

Found in
Android 5.0
Release



Beginning Android Wearables

Andres Calvo

Apress®

www.allitebooks.com

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Apress®

Contents at a Glance

About the Author	xix
About the Technical Reviewer	xxi
Acknowledgments	xxiii
■ Part I: Introduction	1
■ Chapter 1: Introducing Android Wearables	3
■ Part II: Notifications.....	17
■ Chapter 2: Reviewing Notifications for Android	19
■ Chapter 3: Customizing Notifications for Wearables.....	55
■ Part III: Android Wear	93
■ Chapter 4: Running Apps Directly on Android Wear	95
■ Chapter 5: Android Wear User Interface Essentials	129
■ Chapter 6: The Wearable Data Layer API	173
■ Chapter 7: Creating Custom Watch Faces	215

- **Part IV: Google Glass** **273**
- **Chapter 8: Running Apps Directly on Glass** **275**
- **Chapter 9: Glass User Interface Essentials**..... **295**
- **Chapter 10: Gesture and Voice Recognition** **349**
- **Chapter 11: The Camera: Taking Pictures and Recording Video** **381**
- **Part V: Android Wear and Glass** **417**
- **Chapter 12: Location and Orientation**..... **419**
- Index**..... **477**

Part **|**

Introduction

Introducing Android Wearables

We already know that mobile devices have been expanding at such a fast pace that they're practically ubiquitous: we can check our email, send and receive chat messages, or even start video conferences regardless of whether we're at home or on the road. Mobile devices, however, have their limitations: they require your dedicated attention and they occupy at least one hand. Wearables devices address these limitations and have the potential of revolutionizing the way we interact with technology. When building apps for wearables, do not forget that wearable devices are not meant to replace mobile devices, but rather to complement them. To provide the best possible user experience for a given application, build it for the most suitable platform. While complex interactions such as browsing the internet are best suited for mobile devices, wearables excel at quickly displaying glanceable information such as a runner's current time and distance.

In addition to providing a better user experience under certain circumstances, wearables also contain sensors that open new possibilities for contextual awareness. For instance, many wearables contain a heart rate sensor, which can help evaluate the quality of a user's workout and offer suggestions.

In this chapter, we'll elaborate on the benefits of wearables, including their potential for increased contextual awareness and unprecedented user experiences. Then, we'll introduce Android Wear and Google Glass, and we'll talk about their basic user interfaces.

Wearables and Contextual Awareness

People typically place mobile devices in their pockets, purses, or backpacks, whereas by definition wearable devices are always placed on the body. In particular, users wear Glass on their faces and Android Wear devices on their wrists. In a way, wearable devices are more personal than mobiles.

By being closer to you, wearable devices are able to learn more about where you are and what you're doing. For instance, while a mobile device knows what direction it's facing, Glass knows what direction *you* are facing. Android Wear devices can also measure or infer context beyond the reach of mobile devices.

Consider, for example, the BioGlass project from MIT Media Lab's Affective Computing group. BioGlass leverages the accelerometer and gyroscopic sensors of Glass to derive a user's heart rate and respiration rate with an accuracy comparable to the values obtained by traditional vital sensors. Heart beats and respiration elicit subtle movements throughout a user's body, and Glass can detect these movements and use them to infer a user's vitals. This procedure is only possible because Glass is in close contact with a user's body. For more information, see the BioGlass homepage (<http://bioglass.media.mit.edu/>).

The LynxFit Glassware, which is available from the MyGlass store, also demonstrates how wearables can achieve contextual awareness that would not be possible with mobile devices. This Glassware guides users through workouts and tracks their progress. During a workout, it utilizes the sensors of Glass to count exercises as users perform them. Being able to detect exercises in real time and concurrently show users feedback is only possible because of the unique form factor of Glass.

A third example is an alarm app I wrote for Android Wear that makes sure users wake up by checking the watch's sensors to ensure they are standing. If users dismiss the alarm and fall asleep, the app detects a short period of inactivity and restarts the alarm. This app can reliably detect periods of inactivity because of its direct contact with a user's wrist.

Wearables and User Experience

A common misconception is that wearables are intended to replace mobile devices. In reality, wearables complement mobile devices and may even be dependent on them. For instance, Android Wear watches cannot access the internet or download any apps without being paired to a mobile device.

The objective of designers and developers should be to maximize the user experience and to quickly present the right information with minimal user intervention. Certain applications are better suited for wearables than mobiles and vice versa. Starting a timer, for instance, is a faster and cleaner process on Glass than on a mobile. On the other hand, crafting a lengthy reply to an email is manageable on a mobile but tedious on Glass. While there is no exact formula for determining what applications are better suited for wearables, in general, wearables are most effective for simple and short interactions. Additionally, wearables are useful in situations where users cannot easily access mobile devices. For example, consider a Glassware that tracks runs and bicycle rides called Strava, which is available from the MyGlass store. Strava utilizes GPS to measure a user's key stats, such as distance and pace. To view these stats during a run or a ride, users must simply look up to see them on Glass. In contrast, viewing these stats on a mobile phone would be distracting and tedious since users would have to take out their phones during the activity. Holding a phone while riding a bicycle is particularly disruptive since users should have both hands on the handlebars.

Why Android Wearables?

Although this book focuses on Android-based wearables (namely Glass and Android Wear), wearable computers have existed for decades before the introduction of Android. In 1961, for instance, Edward Thorp and Claude Shannon built a wearable computer to help increase the odds of winning at roulette. The user would use switches inside his shoe to initialize the computer and time the rotor and the ball. The computer would then calculate the octant on which to bet and transmit the result to a collaborator's earbud through a radio. The collaborator would make the bets to reduce suspicion, since he would be located in a seat with a poor view of the ball and rotor. For more information, see the article "The Invention of the First Wearable Computer" by Thorp (<https://www.cs.virginia.edu/~evans/thorp.pdf>).

Not only did wearable computers exist before Android wearables, but there are also other emerging wearable computers such as the Meta (<https://www.spaceglasses.com/>).

Why should we learn to build applications for Android wearables instead of other devices? Here are a few reasons:

- Android has matured into a powerful platform that we can leverage while building apps for Android wearables.
- Google is working on the next versions of Glass and Android Wear, so the platforms will only get better with time.
- Glass and Android Wear are relatively new platforms, and the best apps and user experiences are yet to be developed.

Android contains tens of thousands of APIs and has been extensively tested on handhelds. Glass and Android Wear leverage most of these capabilities and provide features such as location, orientation, and Bluetooth right out of the box. These capabilities let you build sophisticated apps that would not otherwise be feasible on new platforms.

What Can Android Wear Do?

There are currently several Android Wear watches available in the market. The Samsung Gear Live and the LG G were the first devices to be released and have a square screens. In contrast, the Moto 360 and the LG G R, which were released shortly after, have round screens (see Figure 1-1).

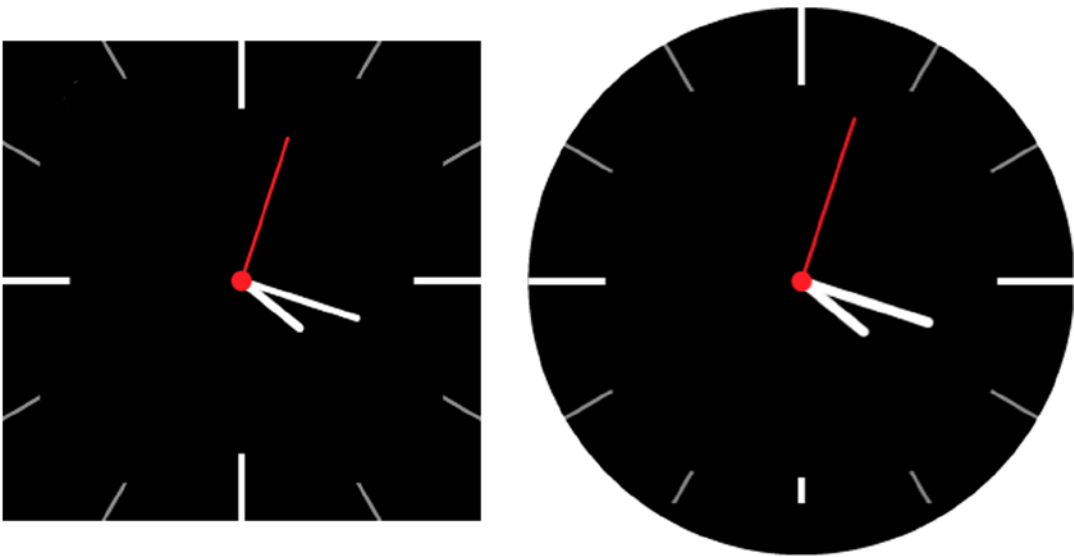


Figure 1-1. Android Wear devices can have square screens (left) or round screens (right)

With regards to hardware, Android Wear devices usually have orientation sensors (that is, an accelerometer, gyroscope, and magnetometer), Bluetooth low energy, a vibrator, a touchscreen, and a microphone, but they vary in terms of other capabilities. Some Android Wear devices have an ambient light sensor, but not all of them. For instance, the Moto 360 has one but both the Samsung Gear Live and the LG G do not. Ambient light sensors are used to automatically adjust a device's brightness depending on current lighting conditions.

Certain watches such as the Samsung Gear Live and the Moto 360 include heart rate sensors, but the LG G does not. These optical sensors measure your heart rate, but are still a little slow and unreliable. Future Android Wear devices will likely improve this sensor and could even contain additional health sensors.

Android Wear devices do typically not have WiFi, a camera, or speakers (although they can pair with bluetooth headphones). Note that Android Wear can still access the internet if paired to a mobile device.

The Context Stream

The interface of Android Wear is based on the context stream, which is a vertically scrolling list of cards similar to a ViewPager. In Android Wear, a card refers to a white card of content, as shown in Figure 1-2. Cards in Android Wear are typically displayed on top of a background image that provides additional context.

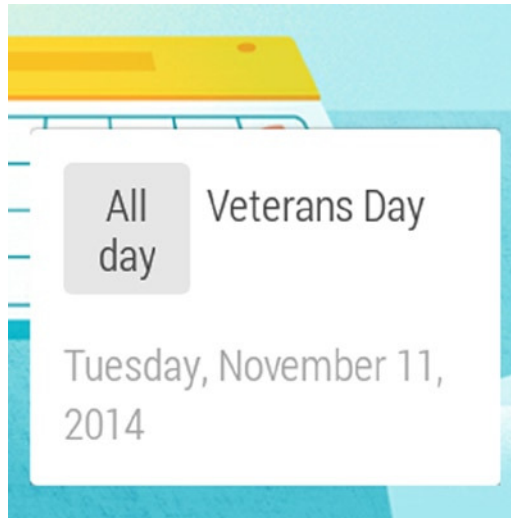


Figure 1-2. A card in Android Wear contains content and is typically displayed on a background that provides additional contextual information

Each card in the context stream can contain additional pages of related information. Users swipe vertically to navigate through the cards in the context stream and swipe horizontally to reveal a particular card's additional pages. When users want to get rid of a card, they dismiss it by swiping from right to left.

Apps can post their own cards to the context stream, which allows users to receive information without having to explicitly open apps. That being said, users can still open apps explicitly by using the cue card.

The Cue Card

Users open the cue card by saying "OK Google" or by tapping on the upper portion of the background of the home screen. The cue card lets users trigger actions by either 1) saying the name of the action out loud or 2) manually selecting the action from a list that can be accessed by swiping up (see the left of Figure 1-3).

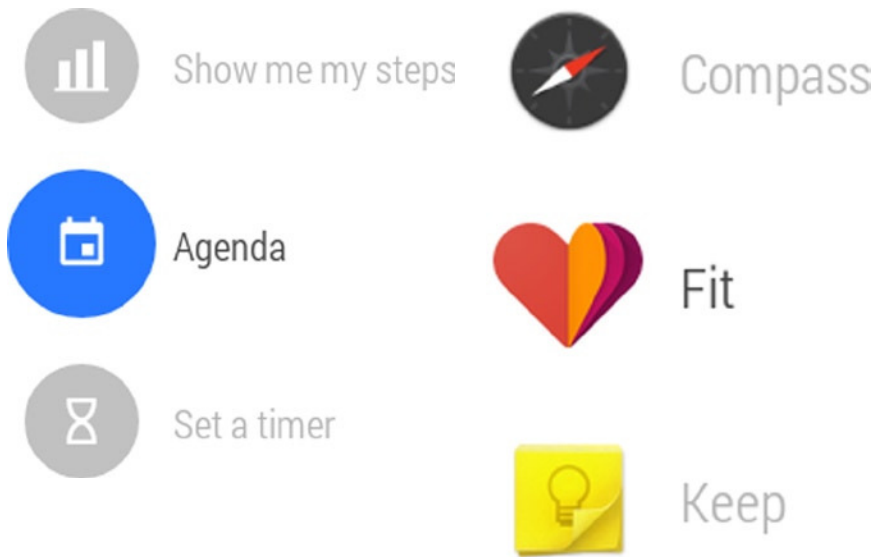


Figure 1-3. The cue card allows users to trigger actions (left). The primary screen displays the default list of actions and (right) custom actions are relegated to a separate menu that can be accessed from the primary screen by tapping on the “Start...” action

Since these actions are usually triggered with voice recognition, they are often called voice actions even though users have the option of triggering them manually. The list of available voice actions is managed by Google and cannot be modified by third party apps. However, apps can listen for the Intents issued by these voice commands and provide alternatives to the system apps.

While third party apps cannot explicitly modify the primary list of actions, they can place custom actions in a separate menu that can be accessed by tapping on the “Start...” action (see the right of Figure 1-3). By relegating custom actions to a separate menu, Android Wear ensures that the primary list of voice actions does not become unmanageable by having too many items.

Android Wear places the most commonly used actions at the top of the cue card. In the event that users frequently access custom commands, these commands will appear in the primary cue card. Note that these custom commands appear in the primary list of actions and seem to contradict the the assertion made in the previous paragraph. However, these custom commands are only shortcuts and will be removed from the primary list if they cease to be frequently used actions.

The Android Wear App

The Android Wear app, which is available on the Google Play store, manages a Bluetooth connection to an Android Wear watch and lets users choose watchfaces and modify settings on their watch. The Bluetooth connection allows the watch to leverage the capabilities of the phone, including internet and location.

Moreover, the Android Wear app synchronizes the mobile device's notifications with the watch. In other words, notifications that are received on the phone are forwarded to the watch. When users dismiss notifications on the watch, they are automatically dismissed on the mobile.

The Android Wear app does not let you download third party apps. Instead, you can download apps directly on the Google Play Store on a handheld device. Handheld apps that are compatible with Android Wear automatically transmit the necessary files over to the watch so that the app resides in both the handheld and Wear. Similarly, when you uninstall an app on your handheld device, any associated Wear apps are automatically uninstalled.

Now that we've covered the interface of Android Wear devices, let's focus on Glass.

What Can Glass Do?

In terms of capabilities, Glass is literally an Android device and thus provides the majority of the capabilities you would find in any other Android device, including Bluetooth Low Energy, WiFi, orientation sensors (that is, an accelerometer, gyroscope, and magnetometer), a camera, a microphone, and an ambient light sensor). Although the current version of Glass (XE 22) is based on Android Kitkat, future version of Glass will likely be based on Lollipop. Additionally, Glass has a small screen that sits above the right eye, a touchpad, and a bone conduction speaker (see Figure 1-4).



Figure 1-4. The screen of Glass sits above the user's eye

Unlike a traditional speaker, a bone conduction speaker does not transmit sound waves directly into the ear. Instead, it creates vibrations on a user's head that are conducted by the skull directly into the inner ear. Listening to a bone conduction speaker sounds exactly the same as listening to a regular speaker, for the most part. To observe a slight difference, try wearing Glass and listening to any sort of audio, and experiment with covering your ears and pressing on the speaker located behind your ear. As a result of bone conduction, you'll notice that both of these actions increase the perceived volume of the audio.

Additionally, Glass has a few rather unexpected sensors. On the other side of the camera, it has an infrared emitter and receiver that are used to detect eye gestures such as blinking. With this sensor, Glass allows users to take pictures by winking. Glass also has a proximity sensor near the infrared sensors that helps Glass implement on-head detection.

Understanding Glass Lingo

Let's go over the terminology typically used to talk about Glass, since it can be confusing to the uninitiated.

Google Glass vs. Glass

Although the device is officially called Google Glass, Google prefers that we refer to it simply as Glass (for more info, see <https://developers.google.com/glass/distribute/branding-guidelines>). My preference is to call it Google Glass only the first time I mention it to prime people to think about Glass the device instead of glass the material.

Glassware

Apps for Glass are officially called Glassware, but people often refer to them as "apps" or "apps for Glass."

Glass Explorers

Google's annual developer conference is called Google IO. Google IO 2012 attendees located in the United States were given the opportunity to become an early adopter and purchase Glass for \$1500. Google referred to early adopters of Glass as Glass Explorers.

In early 2013, Google invited people to answer the question "What would you do if you had Glass?" with the hashtag #ifihadglass to join the Glass Explorers. While winning the ability to purchase a product doesn't sound like a good deal, Google's intent was likely to make sure only people who are interested in advancing the technology were able to purchase the device. After #ifihadglass was over, people could join the Google Explorers program only by receiving invitations from existing Explorers. Then, as of April 2014, Google allowed anyone to purchase Glass and join Glass Explorers without an invitation. In short, the Explorers program is a way of making sure people realize that Glass is still under beta and is going through rapid development.

In January 2015, Google ended the Glass Explorers program and people are no longer able to purchase Glass in its current form. However, there's still a lot we can learn by building Glassware, and in doing so we'll be better prepared for future versions of Glass. We'll elaborate on the end of the Explorers program later this chapter.

Cards & Timeline

The interface of Glass is centered around the timeline, which is a horizontally scrolling list of views similar to a ViewPager. An intuitive way of thinking about the timeline is as a big row of cards that wraps around your head, which is why views that comprise full screen layouts in Glass are referred to as “cards.”

By default, the screen is turned off until users tap on the touchpad or tilt their heads up, the latter of which is called “head wake up” and can be disabled from the settings screen. As soon as the Glass display turns on, you'll see the home screen, which displays the clock and provides access to all the voice commands that start Glassware (see Figure 1-5).



Figure 1-5. The home screen appears as soon as a user turns on the Glass display

The home screen also separates the timeline into two sections: 1) the “past section” consists of all the cards located to the right of the home screen and 2) the “present and future section” consists of all the cards located to the left of the home screen.

The Past Section

The past section of the timeline is comprised of all the cards located to the right of the home screen. Although cards in the past section should be relevant to the user at the time of delivery, they aren't necessarily relevant beyond that point. Moreover, there should be no consequence if a user completely ignores a card from this section. Receiving an email, for instance, appears as a card in the past section of the timeline since it's only important at the time of delivery.

As new cards are added to the past section of the timeline, they are placed closer to the home screen and older cards begin to automatically disappear after seven days, or when there are 200 or more newer cards. Users don't need to "manage" the cards in this section or keep it clean, but rather they should rely on the system to eliminate old ones.

The Present and Future Section

The cards to the left of the timeline comprise the present and future section and contain information that is immediately relevant to the user and should be easy to access. For instance, calendar events are placed in this section of the timeline since they're relevant in a user's short term future. Once a calendar event, or any card in the present and future section for that matter, ceases to be relevant, it should be removed from this section of the timeline. Another example of Glassware that resides in the present and future section is Strava, which tracks runs and bicycle rides and displays key statistics on the screen. Once users complete a run or a ride, real time statistics are no longer relevant and the card that contains the statistics is removed from the present and future section.

Voice Commands

When Glass is displaying the home screen, users can either say "OK Glass" or tap on the touchpad to reveal a list of voice commands that can be used to start Glassware. As new Glassware is installed, additional voice commands appear on the list. Glass automatically rearranges the command list and places the most frequently used commands at the top.

The MyGlass App

While Glass is great at small and simple interactions, more complicated tasks are better suited for mobile devices or computers. For instance, the initial configuration of Glass requires users to type a WiFi password and cannot be performed exclusively on Glass. This configuration can be set by accessing google.com/myglass or by downloading the MyGlass app, which is available on the Google Play Store for Android devices and the Apple App Store for iOS devices.

In addition to setting up Glass, the MyGlass app also allows users to browse and install Glassware. Once users grant Glassware the permissions it needs, Glass automatically downloads the apk (if needed).

The MyGlass app also manages the connection between Glass and the phone. Although Glass can still perform the majority of its functions itself, pairing it to a phone lets Glass leverage the phone's capabilities, including GPS, SMS messaging, and voice calls.

Notification Sync

As of software version XE22, which was released in Oct 2014, Glass synchronizes the notifications of a paired handheld device with the timeline. When you receive a notification on your handheld, it creates a timeline item that displays the same information on Glass. If you trigger a notification action from the timeline item, the action will be performed just as if you had triggered it from the handheld. If you dismiss the notification on Glass, it will also be dismissed on the handheld. We'll thoroughly discuss notification sync in Chapter 3.

The End of the Glass Explorers Program

In January 2015, Google ended the Glass Explorers program, and as a result, people are no longer able to purchase Glass in its current form. Additionally, the existing version of Glass will not have any more software updates after XE 22. Although the Glass Explorers program has ended, Glass still has a bright future: Glass is no longer part of Google X, which is a research division, and is now an independent division overseen by the founder of Nest, Tony Fadell. Google is clearly working on the next version of Glass and they even posted new job openings for the Glass team (see <http://glassalmanac.com/google-posts-8-new-jobs-on-the-google-glass-team/7019/>).

Even though the current version of Glass will become outdated as soon as the next version comes out, designing and developing Glassware for the current version of Glass has many benefits. First, it helps us prepare for future versions of Glass since these platforms will likely share many similar traits. Additionally, the experience we gain with wearable devices today will be the foundation on which we build wearable devices tomorrow. When the next version of Glass comes out, we may have to update our existing Glassware, but the updates will certainly not be as extensive as a rewrite.

Furthermore, the current version of Glass is still alive with the Glass at Work program, which focuses on supporting enterprise applications of Glass. For example, Augmedix is developing Glassware that gives physicians access to patient information.

Google has encouraged developers to continue building Glassware, and they have reiterated that the MyGlass store will remain open. Moreover, they indicated that you may still submit Glassware for review and that it will still be added to the MyGlass store if approved.

Should I Develop for Glass or Android Wear?

I recommend learning how to design and develop for both platforms to better understand their benefits and detriments. The camera of Glass, for example, provides opportunities for innovative applications that are beyond the scope of Android Wear. Similarly, Android Wear's vibrator and touchscreen are not available on Glass. Lifelogging and photography applications would be great for Glass, while sleep trackers would be great for Android Wear.

Start by learning to build apps for the device you own. If you have neither Glass nor Wear, start by learning Android Wear, because emulators are available for this platform and not for Glass.

Design Matters

I'll conclude by emphasizing the importance of design for building effective wearable apps. While most desktop software and some mobile apps are themselves a user's primary task, wearable apps rarely take all of a user's attention. When writing a document in Microsoft Word or Google Docs, for example, your primary task is to write the document, and the world outside of your device becomes secondary. With wearables, in stark contrast, the world outside of your device is the primary focus of attention while the wearable itself becomes secondary. As Google's designers put it, "the world is the experience"—take a look at Google IO 2014's "Designing for Wearables" session, which is available on youtube, for more information.

Wearables should thus be unobtrusive, and they should not take a user's attention away from the primary focus of his or her experience, which is the world. An app that tracks runs and displays statistics would be distracting if it displayed too much information. Strava is a successful app because it displays only the fundamental information (such as pace and distance) with a very large font that is easy to see with a quick glance. The line between designers and developers is becoming blurred since fulfilling this need demands the expertise of both groups. If you are a developer, you'll have to learn about design to craft effective apps for wearables. Design will therefore be a recurring theme in this book and the emphasis of Chapter 13.

Reading This Book

The rest of this book will show you how to implement apps for Android Wear and Glass. Each chapter will typically introduce a topic and then show you how to implement it with complete example applications. When learning any topic, I recommend that you first read its introduction. Then, you should clearly understand what each example is trying to accomplish before tackling the sample code. Next, you can either attempt to recreate each example by typing in everything yourself or by following along with the example code, which can be downloaded from the Apress website (<http://www.apress.com/9781484205181>, under the Source Code/Downloads tab) or from the book's Github repository (<https://github.com/AlephNullo/androidwearables>).

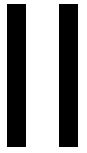
After reading the book and understanding the example code, begin tweaking each app and modifying parts of the implementation at your discretion. Once you feel comfortable manipulating different aspects of the code, try to write a new and related example. You will only become comfortable developing code if you practice writing it yourself and don't just follow along.

The examples in this book will focus on using the APIs from the Android Wear SDK and the GDK. I am assuming that you are able to write basic Android applications and that you are able to write activities and services for handheld devices. If you do not have this prerequisite knowledge, I recommend learning the fundamentals before continuing. There are many great resources available, including the official documentation (see <https://developer.android.com/training/>). You might also want to check out another Apress book, Wallace Jackson's *Android Apps for Absolute Beginners* (<http://www.apress.com/9781484200209>).

Summary

We learned that Android wearables—that is, Android Wear and Glass—provide a unique opportunity to build innovative apps. We then covered the basic interface and terminology associated with each device and concluded by emphasizing the importance of good design. The next chapter reviews notifications on Android handhelds, which will help us understand how to build notifications for wearables in Chapter 3.

Part



Notifications

Reviewing Notifications for Android

Notifications provide users with relevant and timely information even when the user is not actively using your app. For instance, they notify the user of asynchronous events such as the arrival of an email or the completion of a download. They also provide contextually relevant information, such as reminding you to carry an umbrella on a rainy day.

The Android Wear platform automatically shares notifications between connected handhelds and wearables. That is, notifications issued on handhelds automatically appear on wearables without requiring any additional code. While we can customize notifications to display information more effectively on wearables (see Chapter 3), it is essential to understand how to craft notifications on handhelds first.

We'll start this chapter by reviewing how to build basic notifications. Then, we'll go over the ways in which Android 5.0 (also known as Lollipop) improved notification visibility and accessibility, including lock screen notifications and heads-up notifications. Afterwards, we'll dive into rich notifications, and we'll build notifications that allow users to press a button on a notification to trigger an action, such as playing or pausing a song. Finally, we'll see how to craft notifications with custom layouts, and we'll build a rich notification using `MediaStyle`, which was introduced in Android 5.0.

The Example App

This chapter contains several examples that demonstrate how to build different types of notifications. The sample source code is located in the `app` module of the `HandheldNotifications` project. The user interface for `MainActivity` (see Figure 2-1) contains several buttons that trigger the notifications covered in this chapter.

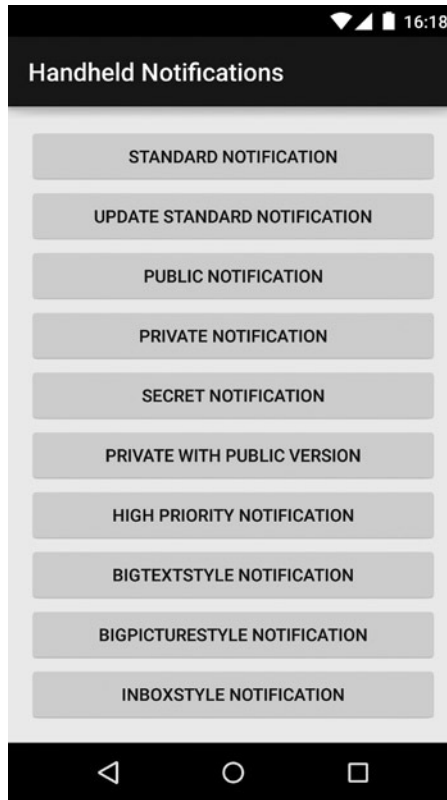


Figure 2-1. MainActivity contains several buttons that trigger the notifications covered in this chapter. There are additional buttons that can be accessed by swiping up. They are not shown in the screenshot

This section outlines the implementation of MainActivity.

1. Create a layout.

The layout of MainActivity is wrapped by ScrollView to ensure that users can access all buttons on devices with small screens.

In res ► layout ► activity_main.xml:

```
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:padding="16dip"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
```

```
<Button
    android:text="Standard Notification"
    android:onClick="onStandardNotificationButtonClick"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />

<Button
    android:text="Update Standard Notification"
    android:onClick="onUpdateStandardNotificationButtonClick"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
...
</LinearLayout>
</ScrollView>
```

The complete layout contains additional buttons that are not shown here for brevity.

2. Declare the activity.

Declare `MainActivity` as a launcher activity and force it to remain in portrait mode. Most apps should handle both portrait and landscape mode, but `MainActivity` only considers portrait mode for simplicity since its orientation does not affect notifications.

In `AndroidManifest.xml`:

```
<activity
    android:name=".MainActivity"
    android:label="Handheld Notifications"
    android:screenOrientation="portrait">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3. Implement the activity.

The `MediaStyle` notification, which is covered at the end of this chapter, requires Android 5.0. In the event that the app runs on a lower version of Android, this button should be disabled. Note that throughout this book, we'll extend the support library's `ActionBarActivity` instead of a regular `Activity` so our examples appear correctly in Android 5.0. If you don't know how to add the `appcompat v7` library, which contains `ActionBarActivity`, see the guide at <http://developer.android.com/tools/support-library/setup.html>.

In `MainActivity.java`:

```
public class MainActivity extends ActionBarActivity {
    public static final int NOTIFICATION_ID = 1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

```

        // disable media style example if running less than API 21
        if(Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
            findViewById(R.id.media_style).setEnabled(false);
        }
    }

    public void onStandardNotificationButtonClick(View view) {
        ...
    }

    public void onUpdateStandardNotificationButtonClick(View view) {
        ...
    }

    ...
}

```

When users click on one of the buttons in this activity, Android will call methods such as `onStandardNotificationButtonClicked`. These methods are implemented throughout the rest of the chapter and are omitted from this code example.

Standard Notifications

Consider an app that notifies you if a flight has been delayed or cancelled. Knowing that a flight is no longer on time can save you a trip to the airport, regardless of whether you're travelling or picking someone up. This flight status notification is only useful if it arrives on time; if it arrives too late, you may already be at the airport.

The most important information in a flight status notification is whether a flight is on time or delayed. That information should therefore appear first in the notification and should stand out in some way. Additionally, the notification should contain the expected departure/arrival time. Even if a flight is on time, the notification is helpful because it provides peace of mind.

The notification in Figure 2-2 contains all of this information in the three fields that all notifications require:

- **Content title:** On time ATL–SFO
- **Content text:** DL1234 departing 6:18pm
- **Small icon:** A white icon of a watch. Although this icon should be related to the content of the notification, we'll use this watch icon throughout this book for simplicity.

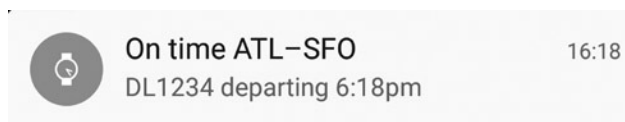


Figure 2-2. A flight status notification indicates that the flight from Atlanta to San Francisco is on time

Note The majority of the screenshots in this chapter were taken on a Nexus 5 running Android Lollipop. The same notifications may look different on your device.

Try to issue notifications when you think they are most useful for the user. For example, a flight status notification should be triggered one to two hours before a flight arrives or two to three hours before a flight departs (and even longer for international flights).

Always Use a Content Intent

Something should always happen when you tap on a notification. If a notification provides no feedback in response to a tap, the app will seem unresponsive. Even if your app is efficient and well written, users will still perceive that it's slow and unresponsive. Most notifications respond to a tap by opening the app that created them. The flight status notification, in particular, should open the main app in response to a tap.

A notification's content intent specifies what action occurs in response to a tap. The action is specified as a `PendingIntent`.

PendingIntents

`PendingIntents` allow other components and applications to start an `Activity`, `Service`, or `BroadcastReceiver` that belongs to your application. `PendingIntents` can be triggered at any time, even if your application has been closed. Moreover, passing a `PendingIntent` to another application gives it permission to start a component that it may not be able to access otherwise.

Android keeps a list of all the `PendingIntents` that have been created and triggers them in response to an action or event. Every `PendingIntent` in the list must be uniquely identifiable in case your app needs to update or cancel a `PendingIntent`. Two `PendingIntents` are equal if their action, data, type, class, and categories are the same. Note that this comparison does not take into account any extra data in the intents. Android considers two `PendingIntents` that only differ in their extras to be the same.

For example, creating a `PendingIntent` that starts an activity has the following form:

```
Intent activityIntent = new Intent(context, ActivityToStart.class);
PendingIntent activityPendingIntent = PendingIntent.getActivity(context, 0, activityIntent, 0);
```

Where the `PendingIntent.getActivity` method accepts four parameters:

- **Context context:** self-explanatory.
- **int requestCode:** an integer that lets you differentiate between `PendingIntents` that would otherwise be identical. That is, if you make two `PendingIntents` that have the same action, data, type, class, and categories but different extras, giving them different request codes would prevent the second `PendingIntent` from overwriting the first.

- **Intent intent:** the intent that will be started when the PendingIntent is triggered.
- **int flags:** specifies what to do when an equal PendingIntent already exists.

When the flags parameter has a value of `PendingIntent.FLAG_UPDATE_CURRENT`, Android retrieves the existing `PendingIntent`, updates its extras, and leaves it in the list. A flag of `PendingIntent.FLAG_CANCEL_CURRENT` tells Android to cancel the existing `PendingIntent` and create a new one. The difference is subtle and doesn't make much difference if you're only creating the `PendingIntent` once or if you don't have extras. Most examples in this chapter would work with either flag.

A Single Top Activity

Consider the case in which users may click on the notification when your app is already in the foreground. Normally, this would start the same activity again, resulting in two instances of the same activity. To avoid this issue, start the activity with the single top flag, which ensures that only a single instance of the activity can be at the top of the task stack.

The flight status notification, which is implemented below, demonstrates how to build a basic notification that includes a content intent that starts an activity with the single top flag.

Implementation

Let's implement a minimal standard notification.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the first button (which is labeled Standard Notification) to trigger the notification.

1. Create a helper method that returns a `PendingIntent` that starts `MainActivity` with the single top flag.

In `MainActivity.java`:

```
private PendingIntent getActivityPendingIntent() {
    Intent activityIntent = new Intent(this, MainActivity.class);
    activityIntent.addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);
    return PendingIntent.getActivity(this, 0, activityIntent,
        PendingIntent.FLAG_UPDATE_CURRENT);
}
```

Place the code from steps 2–4 where you want to issue the notification. The sample code calls this code in the `onStandardNotificationButtonClick` method of `MainActivity.java`.

2. Build a `Notification` and set its content title, content text, and small icon. Always use `NotificationCompat.Builder` to create notifications.

```
PendingIntent activityPendingIntent = getActivityPendingIntent();

Notification standardNotification = new NotificationCompat.Builder(this)
    .setContentTitle("On time ATL-SFO")
    .setContentText("DL1234 departing 6:18pm")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setContentIntent(activityPendingIntent)
    .build();
```

`NotificationCompat.Builder` is part of the Android Support Library and has a minimum API level of 4 while `Notification.Builder` is a part of Android and has a minimum API level of 11. By now, there is little benefit in supporting API levels below 16, so compatibility with previous API levels is not a good enough reason to use `NotificationCompat`. However, `NotificationCompat` contains APIs that let us customize notifications for wearables (as we'll see next chapter), so we'll get into the habit of always using `NotificationCompat.Builder` even though this chapter focuses on notifications for handhelds.

Note `NotificationCompat.Builder` uses the Builder design pattern. If you haven't heard of design patterns, I highly recommend learning about them because they will drastically improve the way you code. The quintessential reference is the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (typically referred to as the Gang of Four) (Addison-Wesley Professional, 1994). You can find more information on the builder pattern at this link: <http://www.oodesign.com/builder-pattern.html>.

3. Obtain an instance of `NotificationManagerCompat`.

```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
```

4. Finally, issue the notification.

```
notificationManager.notify(NOTIFICATION_ID, standardNotification);
```

The notification ID uniquely identifies a notification. If you create multiple notifications with the same ID, the last notification will overwrite previous ones.

Updating Notifications

Notifications should contain relevant, up to date, information. A flight status app should update the flight status notification if a plane gets delayed after you display the initial “on time” notification (see Figure 2-3).

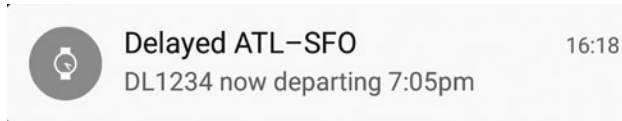


Figure 2-3. The updated notification shows that the flight is now delayed

In this section, you’ll learn how to update a notification.

Implementation

You can update an existing notification by issuing a new notification with the same notification ID.

Note With the example app running, tap the first button (which is labeled *Standard Notification*) to trigger the original notification. To update this notification and show that the flight is now delayed, tap the second button (which is labeled *Update Standard Notification*).

Place steps 1–2 where you want to update the notification. The sample project contains this code in the `onUpdateStandardNotificationButtonClick` method.

1. Create the updated notification. Note that the `getActivityPendingIntent` method is from the standard notification sample.

```

PendingIntent activityPendingIntent = getActivityPendingIntent();

Notification updatedNotification = new NotificationCompat.Builder(this)
    .setContentTitle("Delayed ATL-SFO")
    .setContentText("DL1234 now departing 7:05pm")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setContentIntent(activityPendingIntent)
    .build();

```

Don’t forget to give the updated notification the same content intent as well as any other parameters that haven’t changed.

2. Issue the updated notification with the same notification ID as the original.

```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, updatedNotification);
```

Using the same notification ID is crucial. If you use a different ID, you will create a second notification instead of updating the first, and you'll see both notifications at the same time.

Notification Priority

Not all notifications are equally important. For instance, a tornado warning notification is significantly more important than a chat notification, which in turn is more important than a shopping deal notification. A notification's *priority* specifies its importance and can take any of these values: `Notification.PRIORITY_MIN`, `Notification.PRIORITY_LOW`, `Notification.PRIORITY_DEFAULT`, `Notification.PRIORITY_HIGH`, or `Notification.PRIORITY_MAX`.

For a thorough description of what each of these priorities means, see the notification design documentation (<http://developer.android.com/design/patterns/notifications.html>). The documentation summarizes the priorities with the diagram shown in Figure 2-4.



Figure 2-4. The significance of each value of notification priority. Diagram taken from the notification design documentation (<http://developer.android.com/design/patterns/notifications.html>)

Specifying an appropriate priority for a notification helps Android take measures to ensure users see the most important notifications first. For example, a device may take into account notification priority while sorting its notifications. Moreover, notifications of higher priority are more likely to be expanded by default.

The heads-up notification section demonstrates how to set a notification's priority.

Notification Alerts

When users are not paying attention to their devices, notifications can play a sound, vibrate the device, or flash the device's LED to prompt users to check their devices.

Note A handheld’s LED is usually located on the front side and can blink at a low frequency to indicate that there are unread notifications.

When you are designing a notification, ask yourself if this notification is important and if the user needs to see it right away. Your notification should have alerts only if the answer is yes.

Certain “daily deal” apps frequently use alerts while issuing notifications for shopping deals. However, shopping deals are really not important enough to warrant alerts, and they tend to annoy users. Users are likely to mute annoying notifications (such as those pesky daily deals) or perhaps even uninstall the app that issues them. Use alerts sparingly and only when absolutely necessary.

Although Android supports custom alert sounds, vibration patterns, and LED blinking patterns, there’s no need to get fancy most of the time. Using the default alert sounds and patterns ensures that the user will recognize the alert as a notification alert. If users hear an unfamiliar sound, they may think that it came from someone else’s phone or that something strange is happening to their device.

Default alerts are not only easy to recognize, they’re also easy to implement. Specify what types of alerts you want to give your notification as a parameter to the `setDefaults` method of `NotificationCompat.Builder`. A few examples will clarify. To specify

- an alert sound, call
`.setDefaults(Notification.DEFAULT_SOUND)`
- a vibration, call
`.setDefaults(Notification.DEFAULT_VIBRATE)`
- blinking the phone’s LED, call
`.setDefaults(Notification.DEFAULT_LIGHTS)`
- an alert sound and vibration, call
`.setDefaults(Notification.DEFAULT_SOUND | Notification.DEFAULT_VIBRATE)`
- an alert sound, vibration, and blinking the LED, call
`.setDefaults(Notification.DEFAULT_ALL)`

For example, a chat message warrants the user’s immediate attention, so using all three alert mechanisms is a good idea in this case. If your notification is not that important or is not time sensitive, you may choose to only blink the phone’s LED, or perhaps not to display any alerts at all. In general, you should use the least obtrusive alert that is reasonable for your notifications. The heads-up notification section demonstrates how to implement a notification alert.

Android Lollipop introduced additional parameters for basic notifications that help the system display notifications more effectively. The next section shows how to craft notifications that take advantage of Android Lollipop.

Notifications in Android 5.0

Android 5.0 displays notifications in a different style and with increased visibility and accessibility than previous versions of Android. Below, we'll go over the most important changes that characterize notifications in Android 5.0.

Dark Content on a Light Background

In previous versions of Android, every notification consisted of light text on a dark background. In contrast, Android 5.0 notifications contain dark text on a light background. When Android 5.0 draws notification icons, it draws them as masks (that is, it only takes into account its alpha channel) and places them on top of a circular gray background. Therefore, properly crafted icons, which should consist of a white foreground on top of a transparent background, should look good regardless of what version of Android a device is running. Most notifications will automatically appear correctly on both Android 5.0 and previous versions automatically (see Figure 2-5).

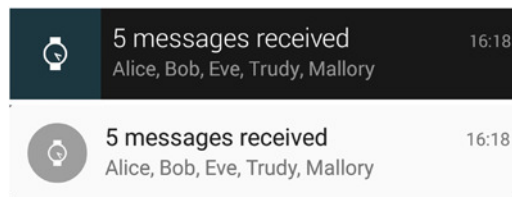


Figure 2-5. Notifications have a different appearance in Android 5.0. Notifications prior to Android 5.0 have light text on a dark background (top). Notifications in Android 5.0 have dark text on a light background (bottom)

The Interruption Filter

As previously covered, notification alerts play a sound, vibrate a device, or blink an LED to indicate that a notification has arrived. If users do not want to be interrupted, they can change a setting called the interruption filter, which determines which notifications are allowed to issue alerts. The interruption filter can take the following values:

- **all:** any notification can issue an alert
- **none:** no notification can issue an alert
- **priority:** only priority interruptions are able to issue alerts. By default, priority interruptions consist of alerts and calendar events, but users can configure what types of notifications constitute priority interruptions.

To change a device's interruption filter, press the volume while viewing the launcher screen and select the value from the tabs located below the volume slider (see Figure 2-6).

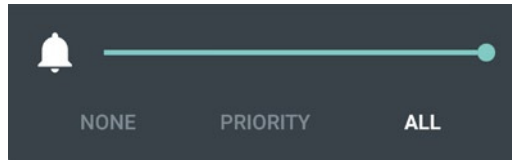


Figure 2-6. In Android 5.0, users can change the interruption filter with the tabs that are located below the volume slider when the volume is changed from the launcher screen

Notification Category

A notification's category, which describes what type of information the notification contains, helps Android sort notifications and allows the interruption filter to filter notifications by type. You should always set a notification's category as a good practice.

Lock Screen Notifications

To view notifications prior to Android 5.0, users had to unlock their devices and open the notification drawer. Android 5.0, however, can display notifications directly on the lock screen as shown in Figure 2-7.

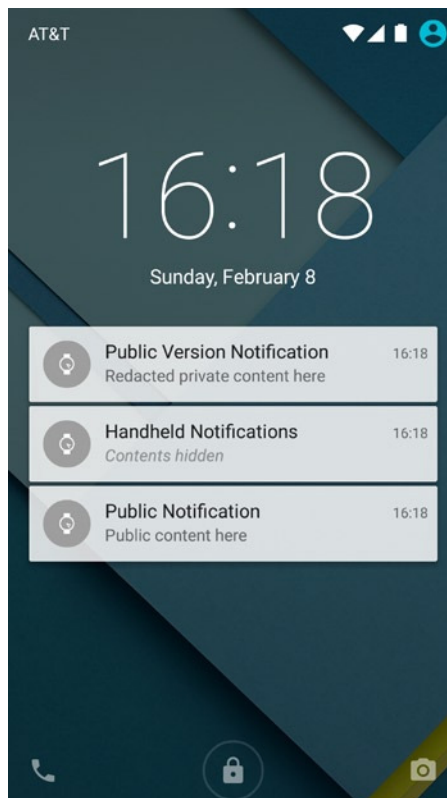


Figure 2-7. With Android 5.0, the lock screen can display notifications

While viewing notifications on the lock screen is convenient, some users may be concerned that other people can read their notifications even if their devices are locked. For this reason, Android allows users to configure how devices display notifications on the lock screen by changing the “when a device is locked” setting, which is contained in the “Sound & notification” group of settings (see Figure 2-8).

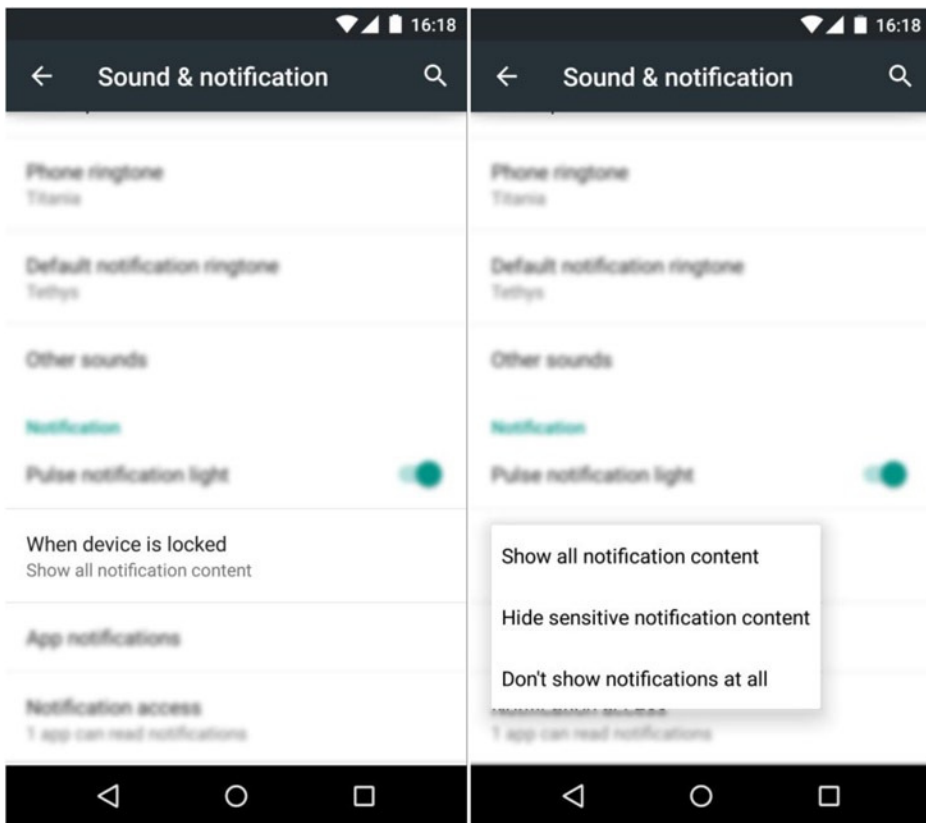


Figure 2-8. The “when device is locked” setting specifies how devices display notifications on the lock screen

In particular, the “when a device is locked” setting can take the following values:

- **show all notification content:** displays all notifications on the lock screen.
- **hide sensitive notification content:** this setting is only available on devices that have a secure lock screen (that is, one that requires a pattern, PIN, or password to unlock). If selected, the lock screen modifies the way certain notifications are displayed, as explained below.
- **don't show notifications at all:** does not show any notifications on the lock screen.

A notification's visibility parameter controls how they are displayed on the lock screen when "hide sensitive notification content" is selected:

- **Notification.VISIBILITY_PUBLIC (default):** shows the entire content of the notification
- **Notification.VISIBILITY_SECRET:** completely hides a notification
- **Notification.VISIBILITY_PRIVATE:** shows a notification but replaces its content with generic or redacted text

Note that notifications with a visibility of secret would still be displayed on the lock screen if "when a device is locked" is set to "show all notification content." That is, while apps can specify the sensitivity of notifications, users have the final say as to how they're displayed on the lock screen.

Implementing Public, Private, and Secret Notifications

This section demonstrates how a notification's visibility affects its appearance on the lock screen.

Note With the example app running, tap on buttons three to five (which are labeled *Public Notification*, *Private Notification*, and *Secret Notification*, respectively) to trigger a public, private and secret notification.

Place the code from steps 1–2 where you want to issue the notification. The sample project calls this code in the `onPublicNotificationClick` method.

1. Create a notification with the desired visibility.

```
Notification notification = new NotificationCompat.Builder(this)
    .setContentTitle("Public Notification")
    .setContentText("Public content here")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setCategory(Notification.CATEGORY_STATUS)
    .setVisibility(Notification.VISIBILITY_PUBLIC)
    .build();
```

2. Issue the notification.

```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, notification);
```

To implement a private or secret notification, simply replace `setVisibility`'s parameter with `Notification.VISIBILITY_PRIVATE` or `Notification.VISIBILITY_SECRET`, respectively. The sample project implements notifications with these visibilities in the `onPrivateNotificationClick` and `onSecretNotificationClick` methods.

Implementing Private Notifications with a Public Version

When “hide sensitive notification content” is selected, the content of private notifications on the lock screen is hidden. In particular, the notification’s content title is replaced with the name of the activity that triggered the notification, and its content text is replaced with the text “Contents hidden.” Notifications can override this default behavior by providing an alternate “public version” that is shown on the lock screen. For instance, a chat notification could provide a public version that indicates how many messages have been received as opposed to showing the content of the messages.

This section implements a private notification with a public version.

Note With the example app running, tap on the sixth button (which is labeled *Private with Public Version*) to trigger a private notification with a public version.

Place the code from steps 1–3 where you want to issue the notification. The sample project calls this code in the `onPrivateNotificationWithPublicVersionClick` method.

1. Create a public version for the notification.

```
Notification publicNotification = new NotificationCompat.Builder(this)
    .setContentTitle("Public Version Notification")
    .setContentText("Redacted private content here")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setCategory(Notification.CATEGORY_STATUS)
    .build();
```

2. Create the private notification and set the notification from step 1 as its public version.

```
Notification notification = new NotificationCompat.Builder(this)
    .setContentTitle("Private Notification")
    .setContentText("Sensitive or private content here")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setCategory(Notification.CATEGORY_STATUS)
    .setVisibility(Notification.VISIBILITY_PRIVATE)
    .setPublicVersion(publicNotification)
    .build();
```

3. Issue the private notification.

```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID+2, notification);
```

For all the notifications that demonstrate visibility, we utilize distinct notification IDs to let us see all notifications at the same time for easier comparison.

Also note that the notification will not play an alert sound if the device is muted. Similarly, the notification will not play any type of alert if the device's interruption filter is set to “none” or “priority”.

Figure 2-9 shows how these notifications appear on the lock screen.



Figure 2-9. The appearance of public, private, and secret notifications on the lock screen. All notifications appear when “show all notification content” is selected (left). No notifications appear when “don’t show notifications at all” is selected (middle). Public and private notifications appear when “hide sensitive notification content” is selected, but the content of the private notification is redacted (right)

Heads-up Notifications

An interruptive priority is either `Notification.PRIORITY_HIGH` or `Notification.MAX`. A notification that has both an interruptive priority and an alert is called an interruptive notification. When users receive an interruptive notification in Android 5.0, it initially appears as a heads-up notification, which is a small floating window that allows users to immediately view and interact with notifications without having to check the notification drawer (see Figure 2-10).

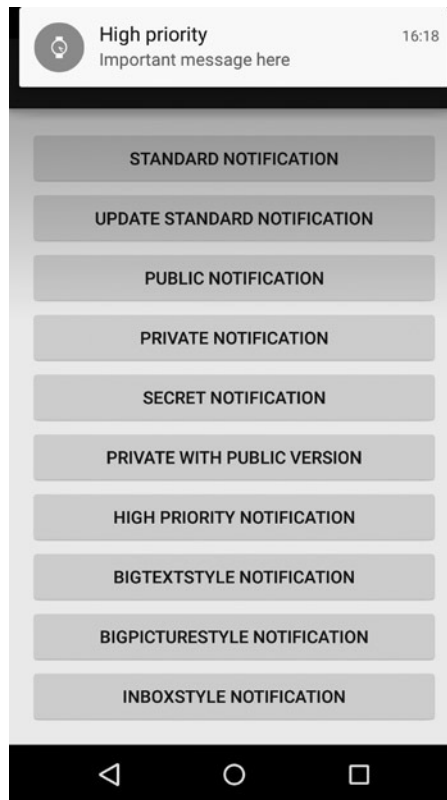


Figure 2-10. A heads-up notification appears as a floating window at the top of the screen

After a short period of time, a heads-up notification disappears, and the notification appears in the notification drawer. Interruptive notifications should be used sparingly because heads-up notifications are more disruptive than traditional notifications.

Implementation

This section demonstrates how to implement a heads-up notification.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the seventh button (which is labeled *High Priority Notification*) to trigger the notification. On devices with Android 5.0, this example also results in a heads-up notification.

Place the code from steps 1–2 where you want to issue the notification. The sample project calls this code in the `onHighPriorityNotificationClick` method.

1. Create a notification with an interruptive priority and an alert.

```
Notification notification = new NotificationCompat.Builder(this)
    .setContentTitle("High priority")
    .setContentText("Important message here")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setPriority(Notification.PRIORITY_HIGH)
    .setDefaults(Notification.DEFAULT_ALL)
    .setCategory(Notification.CATEGORY_STATUS)
    .build();
```

2. Issue the notification.

```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, notification);
```

If you receive this notification when a device is locked, it will also slowly blink its LED. However, since our sample app only issues a notification when the phone isn't locked, we do not see the LED blink. Regardless, the notification still plays a sound and vibrates the device, and it displays a heads-up notification on Android 5.0.

Changing a Notification's Color

By default, notifications in Android 5.0 display a circular gray background behind each notification icon. This background color can be changed using `NotificationCompat`'s `setColor` method.

Compatibility

As long as you utilize `NotificationCompat.Builder` to create notifications, compatibility should not be an issue. Parameters that are only applicable to Android 5.0, such as visibility and category, will simply be ignored by previous versions.

Now that we've covered the nuances of Android 5.0, let's continue discussing notifications that apply to both Android 5.0 and previous versions.

Rich Notifications

Rich notifications, which were introduced with Android Jellybean (API 16), let you create notifications with large content areas, images, and buttons. These rich notification features let you display a lot more information in a notification than you are able to display in a standard notification.

Rich notifications have two states: a normal state and an expanded state. To expand a notification, users must swipe down on the notification with two fingers. Additionally, the notification may appear expanded by default if there is enough available room.

Chat applications such as Hangouts deliver a notification every time you receive one or more messages. While only short messages can be displayed in standard notifications, rich notifications can display messages with long text and pictures.

BigTextStyle Notification

Some chat messages are very long, but a standard notification's content text can only show a single line of text. Before rich notifications were available, the user would have to tap the notification to open the app and see the rest of the chat message. A type of rich notification called a `BigTextStyle` notification shows multiple lines of text when the notification is expanded (see the bottom of Figure 2-11). When it's not expanded, the notification shows the same content as a standard notification (see the top of Figure 2-11).

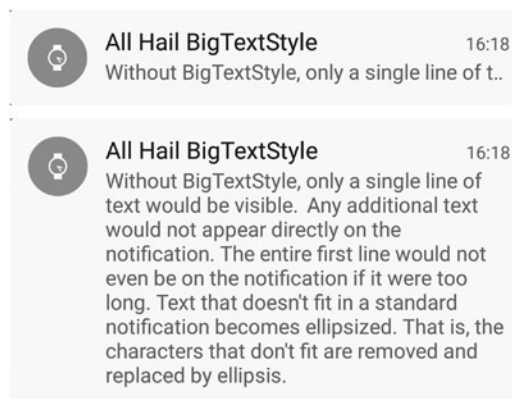


Figure 2-11. A `BigTextStyle` notification as displayed while not expanded (top) or expanded (bottom)

Although opening an app is far from a daunting task, the real advantage of rich notifications in handheld apps is that they can give users a moment of delight. It feels good to save a couple of seconds, especially if it involves a frequently occurring event, such as receiving chat messages.

Implementation

Now that we've seen what a `BigTextStyle` notification looks like, let's implement it.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the eighth button (which is labeled *BigTextStyle Notification*) to trigger the notification.

Place the code from steps 1–3 where you want to issue the notification. The sample project calls this code in the `onBigTextStyleButtonClick` method.

1. Create an instance of `NotificationCompat.BigTextStyle`.

```
String longText = "Without BigTextStyle, only a single line of text would be visible. " +  
    "Any additional text would not appear directly on the notification. " +  
    "The entire first line would not even be on the notification if it were too long! " +  
    "Text that doesn't fit in a standard notification becomes ellipsized. " +  
    "That is, the characters that don't fit are removed and replaced by ellipsis.";
```

```
NotificationCompat.BigTextStyle bigTextStyle = new NotificationCompat.BigTextStyle()  
    .bigText(longText);
```

2. Create a notification and set its style to `bigTextStyle`.

```
Notification bigTextStyleNotification = new NotificationCompat.Builder(this)  
    .setContentTitle("All Hail BigTextStyle")  
    .setContentText(longText)  
    .setSmallIcon(R.drawable.ic_stat_notify)  
    .setContentIntent(activityPendingIntent)  
    .setCategory(Notification.CATEGORY_STATUS)  
    .setStyle(bigTextStyle)  
    .build();
```

3. Issue the notification.

```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);  
notificationManager.notify(NOTIFICATION_ID, bigTextStyleNotification);
```

The notification should be able to stand by itself regardless of whether or not it's expanded. That is, users will not necessarily see both the normal and the expanded notification. Your app should continue to work without any major disruptions no matter which version of the notification the user views.

BigPictureStyle Notification

If you receive a picture as a chat message, you can display it in an expanded notification with `BigPictureStyle`, as shown in the bottom Figure 2-12. If the notification is not expanded, it will look just like a standard notification, as shown at the top of Figure 2-12.

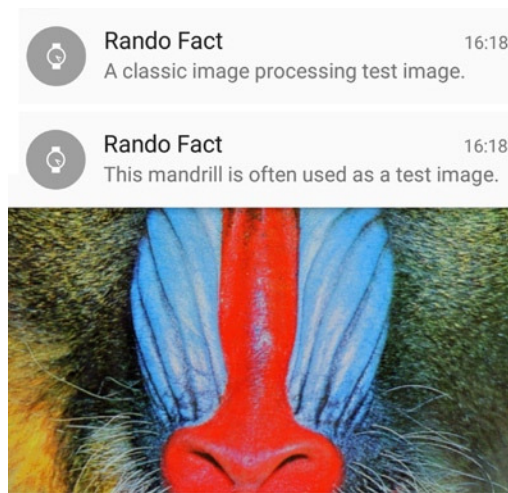


Figure 2-12. A `BigPictureStyle` notification as displayed while not expanded (top) or expanded (bottom). Note that the text underneath the title of the notification is different between the unexpanded and expanded versions. When a notification is expanded, it displays the summary text instead of the content text

Implementation

This section implements a `BigPictureStyle` notification.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the ninth button (which is labeled (which is labeled *BigPictureStyle Notification*) to trigger the notification.

Place the code from steps 1–3 where you want to issue the notification. The sample project calls this code in the `onBigPictureStyleButtonClick` method.

1. Create an instance of `NotificationCompat.BigPictureStyle`.

```
Bitmap bigPicture = BitmapFactory.decodeResource(getResources(), R.drawable.mandrill);
String contentText = "A classic image processing test image.";
String summaryText = "This mandrill is often used as a test image.";
```

```
NotificationCompat.BigPictureStyle bigPictureStyle = new NotificationCompat.BigPictureStyle()
    .setSummaryText(summaryText)
    .bigPicture(bigPicture);
```

The `BigPictureStyle` summary text replaces the content text while the notification is expanded. Also note that the maximum height of the big picture is 256dp.

2. Create a Notification and set its style to `bigPictureStyle`.

```

PendingIntent activityPendingIntent = getActivityPendingIntent();

Notification bigPictureStyleNotification = new NotificationCompat.Builder(this)
    .setContentTitle("Rando Fact")
    .setContentText(contentText)
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setContentIntent(activityPendingIntent)
    .setCategory(Notification.CATEGORY_STATUS)
    .setStyle(bigPictureStyle)
    .build();

```

3. Issue the notification.

```

NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, bigPictureStyleNotification);

```

Once again, the notification should make sense regardless of whether it's expanded or not since not every user will see the expanded state.

InboxStyle Notification

If you receive messages from multiple contacts, you can't preview all of the messages in a single standard notification. At best, you can indicate how many messages you received and perhaps list the senders. With `InboxStyle`, you can preview multiple messages in the expanded notification, one per line (see the bottom of Figure 2-13). If the notification is not expanded, it behaves like a standard notification (see the top of Figure 2-13).

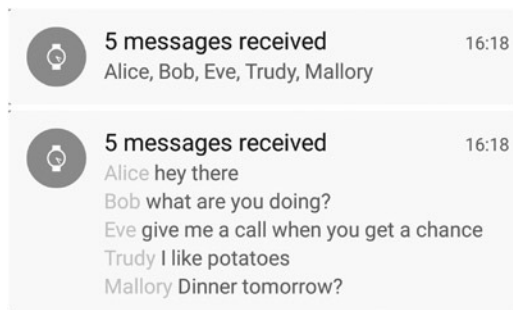


Figure 2-13. An `InboxStyle` notification as displayed while not expanded (top) or expanded (bottom)

Implementation

This section implements an `InboxStyle` notification.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the tenth button (which is labeled *InboxStyle Notification*) to trigger the notification.

1. Create a helper method in `MainActivity.java` that formats a single line of the expanded notification.

```
private Spannable formatInboxStyleLine(String username, String message) {
    Spannable spannable = new SpannableString(username + " " + message);
    int color = getResources().getColor(R.color.notification_title);
    spannable.setSpan(new ForegroundColorSpan(color), 0, username.length(),
        Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
    return spannable;
}
```

A line in the expanded notification is of the form “username message”, where the username is emphasized with a slightly brighter color than the rest of the text. We achieve this effect with a subclass of `CharSequence` called `Spannable`.

`Spannable` lets you apply all sorts of formatting to text. Here, we initialize a `SpannableString` with the username and the message, separated by a space. Then, we apply a `ForegroundColorSpan` to change the color of the username. A `Span` (such as `ForegroundColorSpan`) is applied to a `Spannable` at a certain position and with a given length. In this case, the position is at index zero and the length is the same as the length of the username. This ensures that the `Span` affects only the username.

The `setSpan` method takes a third argument that determines how the span affects text that is inserted after the span is set. Since we don't plan to add any text to the `Spannable`, we don't care about this argument and arbitrarily give it a value of `Spannable.SPAN_EXCLUSIVE_EXCLUSIVE`.

Place steps 2–4 where you want to issue the notification. The sample project runs this code in the `onInboxStyleButtonClick` method.

2. Create an instance of `NotificationCompat.InboxStyle`.

```
NotificationCompat.InboxStyle inboxStyle = new NotificationCompat.InboxStyle()
    .addLine(formatInboxStyleLine("Alice", "hey there"))
    .addLine(formatInboxStyleLine("Bob", "what are you doing?"))
    .addLine(formatInboxStyleLine("Eve", "give me a call when you get a chance"))
    .addLine(formatInboxStyleLine("Trudy", "I like potatoes"))
    .addLine(formatInboxStyleLine("Mallory", "Dinner tomorrow?"));
```

The `addLine` method takes a `CharSequence` to display in the expanded notification. A `CharSequence` is the superclass of `String`, so you could pass a `String` into `addLine`. In this case, however, we want to use the formatted `Spannables` from the first step.

3. Create a Notification and set its style to `inboxStyle`.

```

PendingIntent activityPendingIntent = getActivityPendingIntent();

Notification inboxStyleNotification = new NotificationCompat.Builder(this)
    .setContentTitle("5 messages received")
    .setContentText("Alice, Bob, Eve, Trudy, Mallory")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setPriority(Notification.PRIORITY_HIGH)
    .setContentIntent(activityPendingIntent)
    .setCategory(Notification.CATEGORY_MESSAGE)
    .setStyle(inboxStyle)
    .build();

```

4. Issue the notification.

```

NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, inboxStyleNotification);

```

Don't forget to set the content title and content text to something reasonable because the notification should still make sense if it's not expanded.

Notification Actions

With rich notifications, users can take actions directly on the notification, without having to open the app. In the case of a music player, actions could let you pause, resume, and move back and forth in a playlist. In this sample, these actions are activated by pressing buttons that are only visible in the expanded view, as shown in the bottom of Figure 2-14. The standard view does not display these buttons (see the top of Figure 2-14).

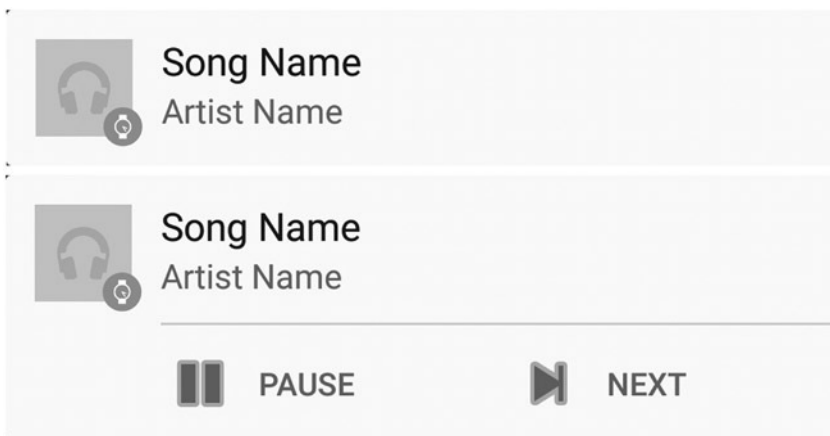


Figure 2-14. A notification with two actions (pause & next) as displayed while not expanded (top) or expanded (bottom)

Additionally, this notification displays a generic album cover instead of a small icon on the left side of the notification. Notifications that have a *large icon* can display images such as avatars or album covers instead of the small icon, which is relegated to the bottom-right of the notification.

Implementation

As you'll see in this section, implementing notification actions is surprisingly straightforward.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the eleventh button (which is labeled *Notification Actions*) to trigger a notification with two actions.

1. If you are writing code from scratch, import the `MediaCommandService.java` file from the sample project.

The notification needs to provide some sort of feedback when the user presses an action button. In a real music app, hearing the music play or pause in response to an action would provide sufficient feedback. However, since this sample does not implement a real music player, we provide feedback for demonstration purposes using text-to-speech and a Toast message.

The `PendingIntents` triggered by an action will start `MediaCommandService`, which will use text-to-speech to say the name of the button the user pressed, after which a Toast will display a message. While text-to-speech is nifty, the focus of this section is on notification actions. Thus, we will not elaborate on the implementation of the service.

2. In `MainActivity.java`, create a helper method that returns a `PendingIntent` that starts the `MediaCommandService` service with a command name as an action.

```
private PendingIntent getMediaCommandPendingIntent(String commandName) {
    Intent intent = new Intent(this, MediaCommandService.class);
    intent.setAction(commandName);
    PendingIntent pendingIntent = PendingIntent.getService(this, 0, intent,
        PendingIntent.FLAG_UPDATE_CURRENT);

    return pendingIntent;
}
```

Place the code from steps 3–6 where you want to issue the notification. The sample project calls this code in the `onNotificationActionsButtonClick` method.

3. Use the helper method from step 2 to create the `PendingIntents` that are triggered by actions.

```
PendingIntent pausePendingIntent =
    getMediaCommandPendingIntent(MediaCommandService.ACTION_PAUSE);
```

```

PendingIntent nextPendingIntent =
    getMediaCommandPendingIntent(MediaCommandService.ACTION_NEXT);

```

When `MediaCommandService` is started, it will look at its `Intent`'s action to determine what action was invoked. It will then say the name of the action out loud with text-to-speech.

4. Obtain the recommended size for a large icon from the system resources and create a `Bitmap` with those dimensions.

```

Resources res = getResources();
int height = (int) res.getDimension(android.R.dimen.notification_large_icon_height);
int width = (int) res.getDimension(android.R.dimen.notification_large_icon_width);
Bitmap largeIcon = BitmapFactory.decodeResource(res, R.drawable.bg_default_album_art);
Bitmap scaledLargeIcon = Bitmap.createScaledBitmap(largeIcon, width, height, false);

```

Before setting a notification's large icon, you must obtain a bitmap of the correct dimensions. These dimensions are usually 64x64dips, but we obtained these numbers directly from the system's resources to avoid hardcoding.

5. Create the `Notification` and set its actions and large icon.

```

PendingIntent activityPendingIntent = getActivityPendingIntent();

Notification actionNotification = new NotificationCompat.Builder(this)
    .setContentTitle("Song Name")
    .setContentText("Artist Name")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setLargeIcon(scaledLargeIcon)
    .setPriority(Notification.PRIORITY_HIGH)
    .setContentIntent(activityPendingIntent)
    .setCategory(Notification.CATEGORY_TRANSPORT)
    .setShowWhen(false)
    .addAction(android.R.drawable.ic_media_pause, "Pause", pausePendingIntent)
    .addAction(android.R.drawable.ic_media_next, "Next", nextPendingIntent)
    .build();

```

Every action takes an icon, a title, and a `PendingIntent`. The icon and the title are used to render the action's button and the `PendingIntent` is triggered when the user presses the button. The `setShowWhen(false)` call ensures that the notification does not display a timestamp since this information isn't relevant for a music player.

6. Issue the notification.

```

NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, actionNotification);

```

While notification actions can be used in a music app, `Play Music` (the best music app, in my opinion) uses a custom notification. We'll implement it in the next section.

Custom Notifications

In versions of Android prior to 5.0, the Play Music app creates a notification like the one in Figure 2-15.

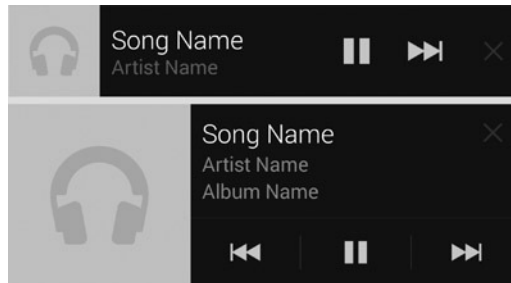


Figure 2-15. A custom notification as displayed while not expanded (top) or expanded (bottom)

This notification has a few interesting features:

- The normal (unexpanded) notification has buttons
- the expanded notification has three lines of text, and
- both versions have a “close” button at the right.

These features cannot be implemented without custom layouts. Regular notifications can’t use custom layouts and they can’t display any actions when unexpanded. Custom notifications are more complicated to implement than regular notifications and should be used sparingly. If you can display all the content you deem essential in regular notifications, then you should not use a custom notification. “But custom layouts can make my notifications stand out, man!” you say. I completely agree. However, when users look at notifications, they won’t be wowed by your out-of-the-box thinking, but rather distracted and annoyed.

Rant aside, if your app truly benefits from custom notifications, then please use them. The Play Music app is a great example. The “close” button is especially useful since you can’t close this notification by swiping it away. Also note that Google decided to include the “previous song” button only in the expanded notification, which has plenty of space, unlike the unexpanded notification. The lack of space isn’t as apparent in Figure 2-15 because the notification has short song and artist names, but typical song names could easily reach the action buttons. Long artist or song names are cut off and ellipsized so that they never overlap the buttons or extend beyond a line.

Implementation

This section implements a custom notification notification.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the twelfth button (which is labeled *Custom Notification*) to trigger the notification.

Place all the following code where you want to issue the notification. The sample project calls this code in the `onCustomNotificationButtonClick` method.

1. Create the layouts for your custom notifications.

The standard (that is, unexpanded) view and the expanded view are two different layouts. In this example, we'll borrow these layouts directly from the Play Music app (and modify them only slightly). You can find them in the book's example source code. The standard layout is in `res > layout-v11 > statusbar.xml` and the expanded layout is in `res > layout-v16 > statusbar_expanded.xml`.

2. Create the `PendingIntents` that are triggered by actions. Note that the `getMediaCommandPendingIntent` method was defined in the notification actions sample.

```
PendingIntent pausePendingIntent =
    getMediaCommandPendingIntent(MediaCommandService.ACTION_PAUSE);
PendingIntent nextPendingIntent =
    getMediaCommandPendingIntent(MediaCommandService.ACTION_NEXT);
PendingIntent prevPendingIntent =
    getMediaCommandPendingIntent(MediaCommandService.ACTION_PREV);
PendingIntent closePendingIntent =
    getMediaCommandPendingIntent(MediaCommandService.ACTION_CLOSE);
```

3. Create an instance of `RemoteViews` and initialize it with the standard layout (that is, the one in `statusbar_expanded.xml`).

```
RemoteViews contentView = new RemoteViews(getApplicationContext().getPackageName(),
    R.layout.statusbar);
contentView.setImageResource(R.id.albumart, R.drawable.bg_default_album_art);
contentView.setTextViewText(R.id.trackname, "Song Name");
contentView.setTextViewText(R.id.artistalbum, "Artist Name");
contentView.setOnClickPendingIntent(R.id.playpause, pausePendingIntent);
contentView.setOnClickPendingIntent(R.id.next, nextPendingIntent);
contentView.setOnClickPendingIntent(R.id.veto, closePendingIntent);
```

`RemoteViews` is essentially a view that can be displayed in another process. This requirement limits the types of views that can be used in layouts for `RemoteViews`. To see a list of the supported views, see <http://developer.android.com/guide/topics/appwidgets/index.html#CreatingLayout>.

4. Create the Notification and set contentView as its content. Note that the getActivityPendingIntent method is taken from the standard notification sample.

```

PendingIntent activityPendingIntent = getActivityPendingIntent();

Notification customNotification = new NotificationCompat.Builder(this)
    .setContentTitle("Song Name")
    .setContentText("Artist Name")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setPriority(Notification.PRIORITY_HIGH)
    .setContentIntent(activityPendingIntent)
    .setCategory(Notification.CATEGORY_TRANSPORT)
    .setContent(contentView)
    .setOngoing(true)
    .build();

```

The call to `setOngoing(true)` prevents the user from swiping the notification away. Since the user is always aware of whether a music app is running (even if it's in the background), an ongoing notification guarantees that the user always gets proper feedback. That is, it guarantees that a notification will always be present when there is music playing. The user can still dismiss the notification by pressing the “close” button. In a real implementation, pressing close would also stop the music.

We'll create the expanded view in the next few steps.

5. In MainActivity, create a helper method that returns true if running Jellybean (API level 16) or above.

```

private boolean isJellybeanOrAbove() {
    return Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN;
}

```

6. Set the expanded content view only if running Jellybean (API 16) or above. Trying to set the expanded view on any previous version of Android would result in a crash.

```

if(isJellybeanOrAbove()) {
    RemoteViews expandedNotificationView =
        new RemoteViews(getApplicationContext().getPackageName(),
            R.layout.statusbar_expanded);
    expandedNotificationView.setImageResource(R.id.albumart,
        R.drawable.bg_default_album_art);
    expandedNotificationView.setTextViewText(R.id.trackname, "Song Name");
    expandedNotificationView.setTextViewText(R.id.artist, "Artist Name");
    expandedNotificationView.setTextViewText(R.id.album, "Album Name");
    expandedNotificationView.setOnClickPendingIntent(R.id.playpause, pausePendingIntent);
    expandedNotificationView.setOnClickPendingIntent(R.id.prev, prevPendingIntent);
    expandedNotificationView.setOnClickPendingIntent(R.id.next, nextPendingIntent);
    expandedNotificationView.setOnClickPendingIntent(R.id.veto, closePendingIntent);

    customNotification.bigContentView = expandedNotificationView;
}

```


Note that the expanded content view must be set directly on the notification and not on the Builder.

7. Issue the notification.

```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, customNotification);
```

The resulting notification looks identical to the one from the Play Music app in versions of Android prior to 5.0. In Android 5.0, however, the notification looks terrible, as shown in Figure 2-16.

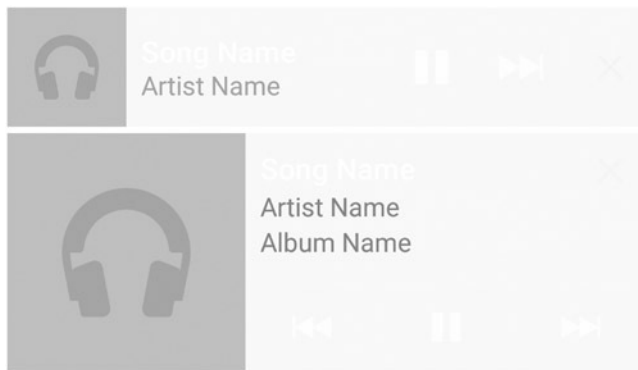


Figure 2-16. A custom notification as displayed in a Nexus 5 running Android 5.0. The unexpanded view (top), and the expanded view (bottom)

In particular, the name of the song and the icons for actions are barely visible because of their low contrast relative to the light background. The names of the artist and album are visible, on the other hand, because their TextViews use the style

```
@android:style/TextAppearance.StatusBar.EventContent
```

This style is provided by the system and defines an appearance of text in notifications that ensures that they are visible. The song name, instead, uses the style

```
@android:style/TextAppearance.StatusBar.EventContent.Title
```

The latter style is also supposed to modify the text appearance to ensure it remains visible, but in this case it's not working as advertised. This could be a bug in Android Lollipop.

To make sure this notification looks good on Android 5.0, we would have to supply another set of action icons and a new style for the appearance for the song name TextView. However, phones running manufacturer skins may not necessarily have a light background. In other words, trying to make the custom notification look good on several devices is challenging. A better alternative is to create a MediaStyle notification for devices running Android 5.0.

MediaStyle Notification

Before Android 5.0, there was no standard way of creating a notification for a media player such as a music app. Notification actions and custom notifications could be used, but their layouts were not standard and could drastically vary from app to app.

Android 5.0 contains a new type of rich notification called `MediaStyle` (see Figure 2-17) that provides a standard appearance for all notifications related to media playback. As of February 2015, the compatibility library does not support `MediaStyle`. To support both Android 5.0 and previous versions, combine `MediaStyle` with one of the approaches highlighted in the previous sections. Note that a `MediaStyle` notification supports up to six action buttons.

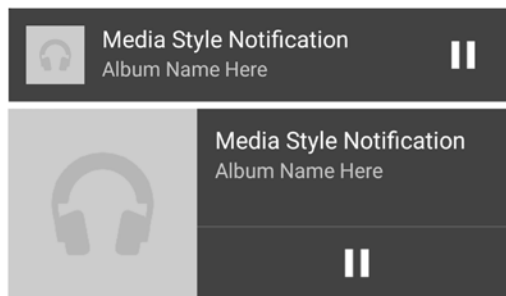


Figure 2-17. A `MediaStyle` notification in Android 5.0. Unlike other notifications, this one has a dark background

The implementation of a `MediaStyle` notification is more involved and resides in a service instead of an activity. We will break the implementation into several pieces.

Implementing `MediaStyleService`

This section implements a service and is responsible for creating the notification and managing music playback.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the last button (which is labeled *MediaStyle Notification*) to trigger a media style notification. This notification will only work on devices running Android 5.0.

Using a service for music playback is essential since the music should not stop when a user leaves an activity. Note that a complete music player app would be significantly more involved than this example, which focuses on demonstrating the use of a media style notification.

Declare and Initialize Member Variables

1. Create the `MediaStyleService` and declare the following member values.

```
@TargetApi(Build.VERSION_CODES.LOLLIPOP)
public class MediaStyleService extends Service {
    public static final int NOTIFICATION_ID = 1;
    public static final String ACTION_PAUSE = "pause";
    public static final String ACTION_PLAY = "play";
    public static final String ACTION_STOP = "stop";
    private MediaPlayer mMediaPlayer;
    private MediaSession mMediaSession;
    private MediaController mMediaController;
    private boolean mPlayingMusic;

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
    ...
}
```

The `@TargetApi` annotation tells Android Studio that we're aware this class uses APIs that are only available in Lollipop. Without this annotation, Android Studio would throw an error since the minimum SDK is lower than Lollipop.

2. Initialize member variables.

The following variables are the most important:

- `MediaSession` registers obtains a unique identifier from the system (that is, the session token) that is associated with media playback. Other classes can then refer to the media playback by using this session token.
- `MediaController` allows us to trigger events such as “play” or “pause” in response to a button press.
- `MediaPlayer` loads a sample mp3 file that is stored as a raw resource.

```
private void init() {
    mMediaSession = new MediaSession(this, "media_session");
    mMediaSession.setCallback(mCallback);

    mMediaController = new MediaController(this, mMediaSession.getSessionToken());

    // play sample mp3
    mMediaPlayer = MediaPlayer.create(this, R.raw.bach_air);
    mMediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
    mMediaPlayer.setOnCompletionListener(mCompletionListener);

    startPlayingMusic();
}
```

Manage Music Playback

When the song is playing, the service should be a foreground service, which makes the notification persistent and prevents users from dismissing it. When the song completes or is paused, the service should no longer be a foreground service to ensure users are able to dismiss the notification.

1. Handle the case in which the song completes. This callback is called by `MediaPlayer`.

```
private MediaPlayer.OnCompletionListener mCompletionListener = new MediaPlayer.
OnCompletionListener() {
    @Override
    public void onCompletion(MediaPlayer mp) {
        setMusicPaused();
    }
};
```

2. Create convenience methods to update the notification when the music is started or paused.

```
private void startPlayingMusic() {
    mMediaPlayer.start();
    mPlayingMusic = true;
    updateMediaNotification();
}
```

```
private void pauseMusic() {
    mMediaPlayer.pause();
    setMusicPaused();
}
```

```
private void setMusicPaused() {
    mPlayingMusic = false;
    stopForeground(false);
    updateMediaNotification();
}
```

3. Update the notification so that it contains a “play” action when the music is paused and a “pause” action when the music is playing. The `buildMediaNotification` method creates a notification with the correct action.

```
private void updateMediaNotification() {
    Notification mediaNotification = buildMediaNotification();

    if(mPlayingMusic) {
        startForeground(NOTIFICATION_ID, mediaNotification);
    } else {
        NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
        notificationManager.notify(NOTIFICATION_ID, buildMediaNotification());
    }
}
```

Create a MediaStyle Notification

The notification's large icon is the album art, and its priority is set to high since the user is constantly listening to the music. Additionally, `setShowWhen(false)` is called to avoid displaying the timestamp of the notification, which is not meaningful for a music player. Also, there is no need to set the notification's category because `MediaStyle` notifications have the correct category (`Notification.CATEGORY_TRANSPORT`) by default.

Note that the notification adds a "pause" action if the music is playing and a "play" action if the music is paused.

1. Implement `buildMediaNotification`.

```
private Notification buildMediaNotification() {
    Bitmap scaledLargeIcon = getAlbumArt();

    Notification.Style mediaStyle = new Notification.MediaStyle()
        .setMediaSession(mMediaSession.getSessionToken())
        .setShowActionsInCompactView(0);

    Intent intent = new Intent(this, MediaStyleService.class );
    intent.setAction(ACTION_STOP);
    PendingIntent pendingIntent = PendingIntent.getService(this, 0, intent, 0);

    // will auto have CATEGORY_TRANSPORT
    // cannot use notificationcompat, alas
    Notification.Builder mediaNotificationBuilder = new Notification.Builder(this)
        .setContentTitle("Media Style Notification")
        .setContentText("Album Name Here")
        .setSmallIcon(R.drawable.ic_stat_notify)
        .setContentIntent(getActivityPendingIntent())
        .setPriority(Notification.PRIORITY_HIGH)
        .setStyle(mediaStyle)
        .setLargeIcon(scaledLargeIcon)
        .setShowWhen(false)
        .setDeleteIntent(pendingIntent);

    if(mPlayingMusic) {
        mediaNotificationBuilder.addAction(makeAction(R.drawable.ic_pause, "Pause",
            ACTION_PAUSE));
    } else {
        mediaNotificationBuilder.addAction(makeAction(R.drawable.ic_play, "Play",
            ACTION_PLAY));
    }

    return mediaNotificationBuilder.build();
}

private PendingIntent getActivityPendingIntent() {
    Intent activityIntent = new Intent(this, MainActivity.class);
    activityIntent.addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);
    return PendingIntent.getActivity(this, 0, activityIntent, PendingIntent.FLAG_UPDATE_CURRENT);
}
```

```
private Bitmap getAlbumArt() {
    Resources res = getResources();
    int height = (int) res.getDimension(android.R.dimen.notification_large_icon_height);
    int width = (int) res.getDimension(android.R.dimen.notification_large_icon_width);
    Bitmap largeIcon = BitmapFactory.decodeResource(res, R.drawable.bg_default_album_art);
    return Bitmap.createScaledBitmap(largeIcon, width, height, false);
}
```

2. When an action is triggered, its `PendingIntent` starts this same service with an intent containing the appropriate action name.

```
private Notification.Action makeAction(int iconResId, String title, String intentAction) {
    Intent intent = new Intent(this, MediaStyleService.class);
    intent.setAction(intentAction);
    PendingIntent pendingIntent = PendingIntent.getService(this, 0, intent, 0);
    return new Notification.Action.Builder(iconResId, title, pendingIntent).build();
}
```

Handle the Play and Pause Actions

1. The `onStartCommand` method handles the `PendingIntents` triggered by the `MediaStyle` notification's actions.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    super.onStartCommand(intent, flags, startId);

    String action = intent.getAction();
    if(ACTION_PAUSE.equals(action)) {
        mMediaController.getTransportControls().pause();
    } else if(ACTION_STOP.equals(action)) {
        mMediaController.getTransportControls().stop();
    } else if(ACTION_PLAY.equals(action)) {
        mMediaController.getTransportControls().play();
    } else if(!mPlayingMusic) {
        init();
        updateMediaNotification();
    }

    return START_STICKY;
}
```

2. When `onStartCommand` receives an action, it forwards it to the `MediaSession` callback through the use of `MediaController`.

```
private MediaSession.Callback mCallback = new MediaSession.Callback() {
    @Override
    public void onPlay() {
        super.onPlay();
        startPlayingMusic();
    }
}
```

```
@Override
public void onPause() {
    super.onPause();
    pauseMusic();
}

@Override
public void onStop() {
    super.onStop();

    mMediaPlayer.stop();
    mMediaPlayer.release();
    mPlayingMusic = false;

    stopSelf();
}
};
```

Now that we've implemented `MediaStyleService`, we must start the service from `MainActivity`.

Implementing `MainActivity`

Place all the following code where you want to issue the notification. The sample project calls this code in the `onMediaStyleNotificationButtonClick` method.

```
public void onMediaStyleNotificationButtonClick(View view) {
    Intent mediaIntent = new Intent(this, MediaStyleService.class);
    startService(mediaIntent);
}
```

At this point, you should be able to test out this example and pause/play music playback.

Summary

We covered all the basics of notifications on Android handhelds, including standard, rich, and custom notifications. We learned about the changes that Android 5.0 imparted on notifications, including lock screen notifications and heads-up notifications. Then, we covered notification actions, which allow users to perform actions directly on the notification without having to open the app. In the next chapter, we'll learn to customize notifications specifically for wearables.

Customizing Notifications for Wearables

Both Android Wear and Glass automatically take the notifications issued on a handheld device and share them with a wearable, as long as both devices are paired. In other words, your handheld apps automatically display notifications on a wearable without having to add any code. However, there are cases in which handheld notifications are not effective on wearables as a result of the different form factor. In these cases, we can customize notifications to behave differently on wearables. To be clear, these customizations do not affect how the notifications are displayed on the handheld but rather they enhance the notifications on wearables.

In Android Wear, each notification issued on the handheld appears as a new card in the context stream. In Glass, each notification appears as a static card in the past section of the timeline. We'll begin this chapter by showing how some of the handheld notifications from Chapter 2 appear on a wearable. All of the notifications from Chapter 2 appear on a wearable without having to change any code, except for the music player notification. Android does not share the music player notification because it uses a custom layout that was only created with a handheld in mind. The rest of the chapter will demonstrate how to create actions that appear only on the handheld or only on the wearable, notifications that span multiple pages, actions that request voice input, and more.

Getting Started

Before reading the rest of the chapter, you should have an Android handheld device paired to Glass or Android Wear. If you don't have a wearable device, use an Android Wear emulator, which can also be paired to a handheld. The official documentation explains how to set up an Android Wear emulator (see <https://developer.android.com/training/wearables/apps/creating.html>).

Then, verify that notification sync is enabled for your wearable device. If notification sync is disabled in Android Wear, a message will appear in the companion app prompting you to “turn on watch notifications.” If using Glass, open the MyGlass app, go to settings, and tap on Notification sync. If disabled, this screen will prompt you to turn on notification sync.

Note Notification sync cannot be enabled on Glass and Android Wear at the same time. Enabling notification sync on Glass disables it in Android Wear and vice versa. Future versions of Glass or Android Wear may fix this issue.

The Example App

This chapter contains several examples that demonstrate how to build different types of notifications. The sample source code is located in the app module of the `WearableNotifications` project. The user interface for `MainActivity` contains several buttons that trigger the notifications covered in this chapter. Consider skipping this section if you already learned about the example app from Chapter 2 since these sections are very similar.

This section outlines the implementation of `MainActivity`.

1. Create a layout.

The layout of `MainActivity` is wrapped by `ScrollView` to ensure that users can access all buttons on devices with small screens.

In `res > layout > activity_main.xml`:

```
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:padding="16dip"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <Button
            android:text="Build Task Stack"
            android:onClick="onBuildTaskStackContentIntentClick"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

        <Button
            android:text="Wearable Only Actions"
            android:onClick="onWearableOnlyActionsClick"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

        ...
    </LinearLayout>
</ScrollView>
```

The complete layout contains additional buttons that are not shown here for brevity.

2. Declare the activity.

Declare `MainActivity` as a launcher activity and force it to remain in portrait mode. Most apps should handle both portrait and landscape mode, but `MainActivity` only considers portrait mode for simplicity since its orientation does not affect notifications.

In `AndroidManifest.xml`:

```
<activity
    android:name=".MainActivity"
    android:label="Wearable Notifications"
    android:screenOrientation="portrait">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3. Implement the activity.

Note that throughout this book, we'll extend the support library's `ActionBarActivity` instead of a regular `Activity` so our examples appear correctly on Android 5.0.

In `MainActivity.java`:

```
public class MainActivity extends ActionBarActivity {
    private static final int NOTIFICATION_ID = 1;
    private static final String GROUP_KEY_MESSAGES = "messages";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void onBuildTaskStackContentIntentClick(View view) {
        ...
    }

    public void onWearableOnlyActionsClick(View view) {
        ...
    }
}
```

When users tap on one of the buttons in this activity, Android will call methods such as `onBuildTaskStackContentIntentClick`. These methods are implemented throughout the rest of the chapter and are omitted from this code example.

Handheld Notifications on Wearables

When a wearable is paired with a handheld, notifications issued on the handheld are automatically shared with the wearable. All of the notifications we wrote in Chapter 2 (except the custom notification) are automatically shared with wearables without changing a line of code, as long as both devices are paired. Custom notifications only appear on the handheld device because the layout created by RemoteViews is not applicable to the wearable.

This section shows how handheld notifications appear on both Android Wear and Glass.

Standard Notifications

In Android Wear, a standard notification displays its content title, content text, and small icon on a single card in the wearable’s context stream. If the notification contains a content intent, it appears as a full screen button to the right of the card. By default, pressing on the button, which is labeled “Open on phone,” displays a brief confirmation animation and triggers the content intent on the handheld (see Figure 3-1). The background color of this notification is automatically selected by Android Wear based on the app icon.

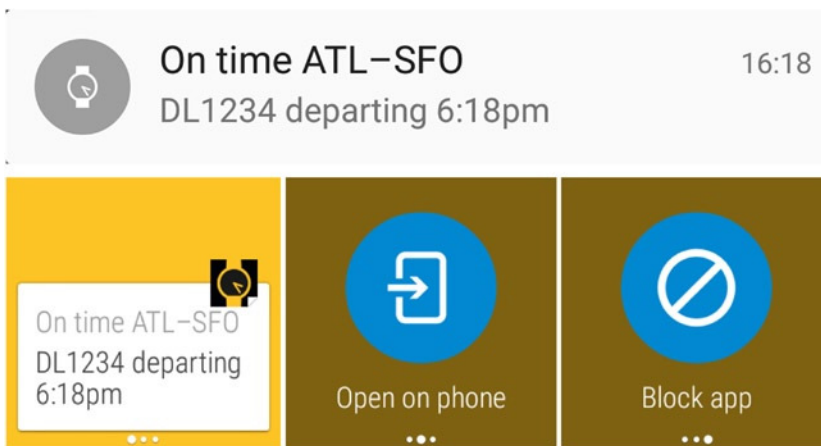


Figure 3-1. A standard notification shared between a handheld and Android Wear. The handheld notification (top). Android Wear displays the notification as a new card in the context stream with three pages. The “Open on phone” action on the second page triggers the content intent on the handheld. The “Block app” action on the third page is automatically inserted by Android Wear and allows users to block undesired notifications. Swiping on the first page from left to right dismisses the notification and removes the card from the context stream (bottom)

In Glass, a standard notification displays its content title, content text, and small icon as a static card in the past section of the timeline (that is, to the right of the home screen). If the notification contains a content intent, it appears as a menu item with a title of “Open on phone.” Selecting this menu item displays a brief confirmation animation and triggers the content intent on the handheld (see Figure 3-2).

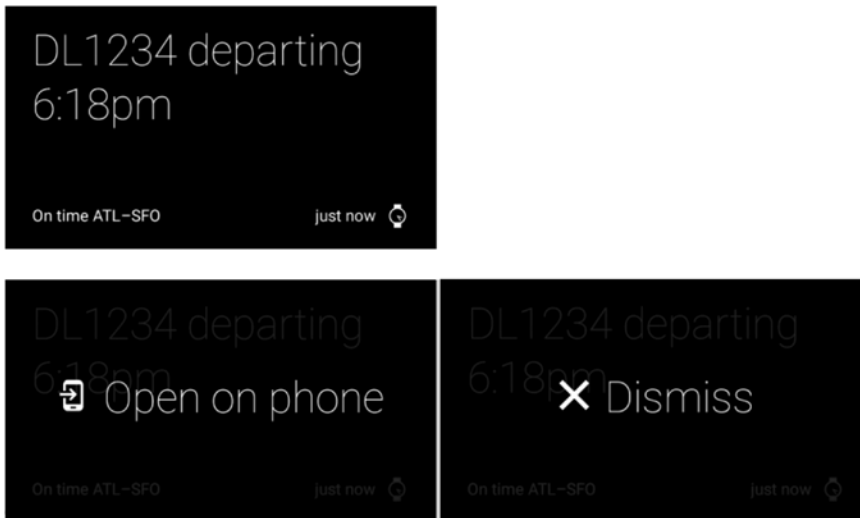


Figure 3-2. A standard notification shared between a handheld and Glass. Glass displays the notification as a static card in the past section of the timeline (top). The card has two menu items: “Open on phone” triggers the content intent on the handheld and “Dismiss” dismisses the notification (bottom)

BigTextStyle Notifications

In Android Wear, `BigTextStyle` notifications appear as an expandable card. Initially, the card shows about four short lines of text, and tapping the card expands it to reveal the rest of the text in a long and scrollable card (see Figure 3-3).

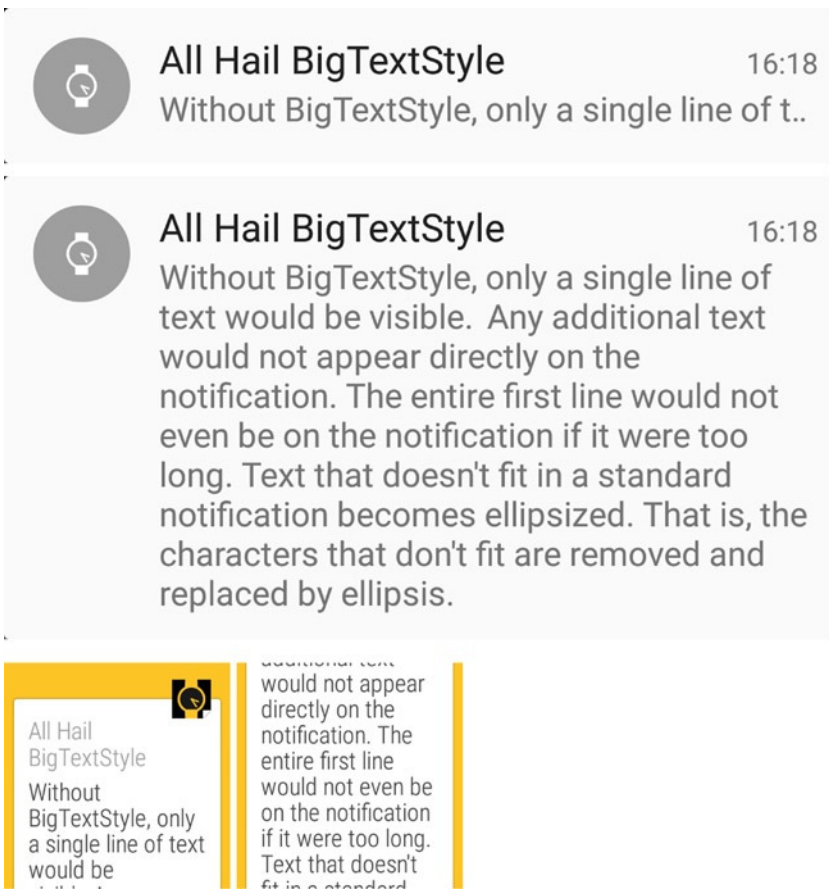


Figure 3-3. A `BigTextStyle` notification shared between a handheld and Android Wear. The unexpanded view in the handheld (top). The expanded view on the handheld (middle). When users tap on the notification in Android Wear, the card expands to reveal additional text in a scrollable card (bottom)

In Glass, `BigTextStyle` notifications appear as a static card in the past section of the timeline. Initially, the card shows about five lines of text, and the rest of the text can be accessed by tapping on the “Show more” menu item (see Figure 3-4).

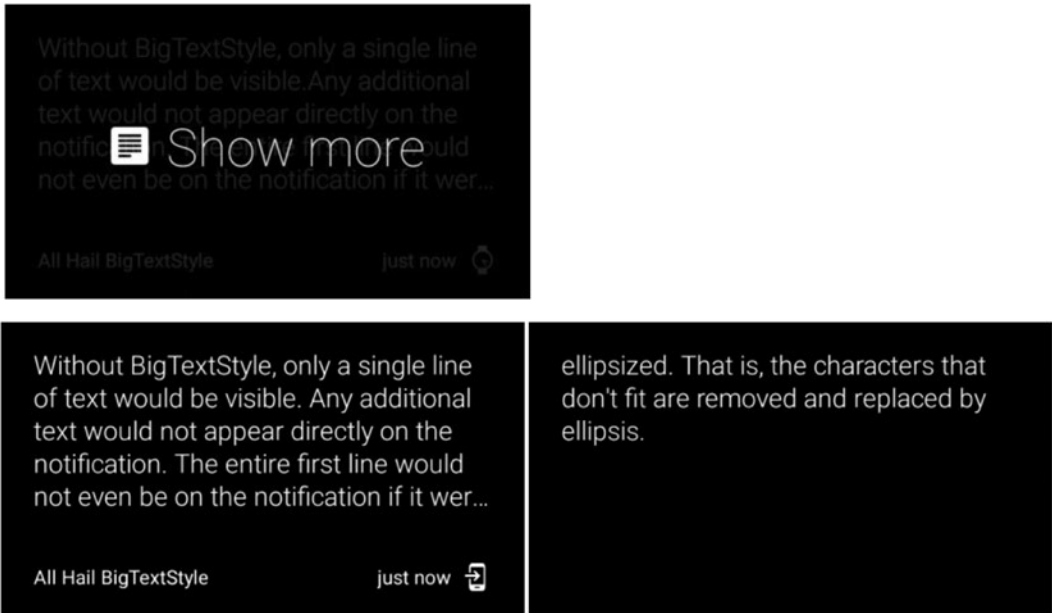


Figure 3-4. A `BigTextStyle` notification shared between a handheld and Glass. The notification shows as much text as can fit in the screen. A menu item called “show more” allows users to view the rest of the screen (top). Selecting the “show more” menu item displays a scrolling list of cards that contains the entire text (bottom)

BigPictureStyle Notifications

In Android Wear, `BigPictureStyle` notifications display the big picture as the background of a card. The main page of this notification shows the content title and text while the second page contains nothing but the image (see Figure 3-5).

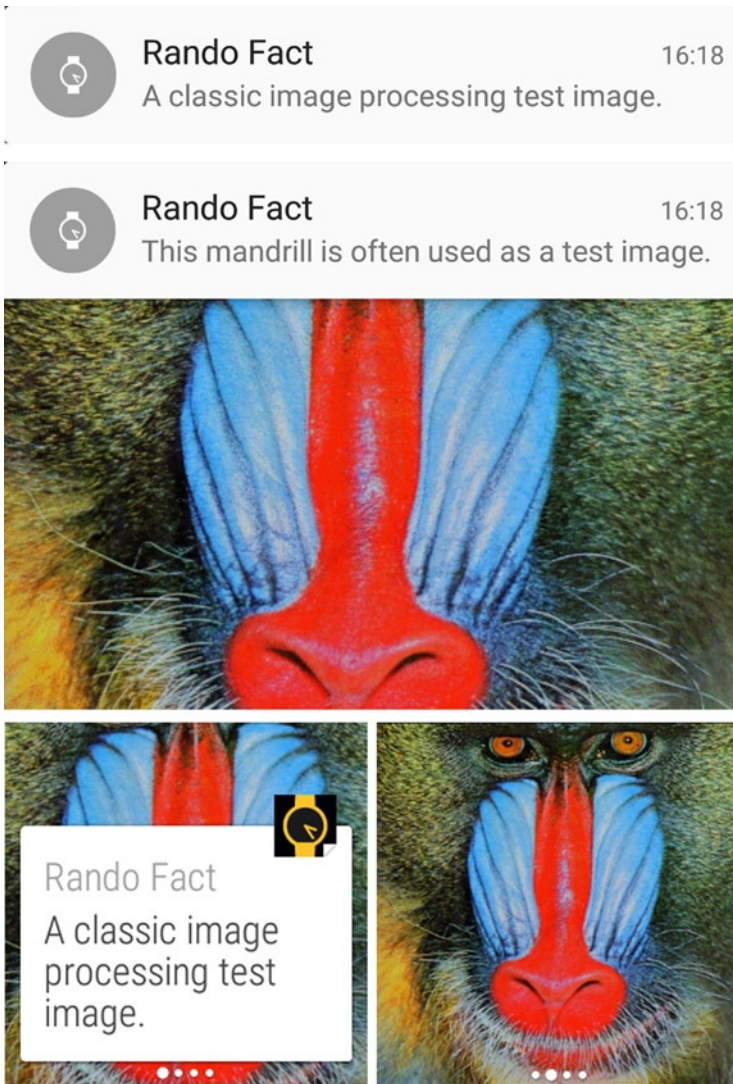


Figure 3-5. A `BigPictureStyle` notification shared between a handheld and Android Wear. The unexpanded notification on the handheld (top). The expanded notification on the handheld (middle). In Android Wear, the first page of the notification displays basic content and the second page only displays the image (bottom)

In Glass, `BigPictureStyle` notifications display the big picture as the background of a card that displays the content title and text as a caption (see Figure 3-6).

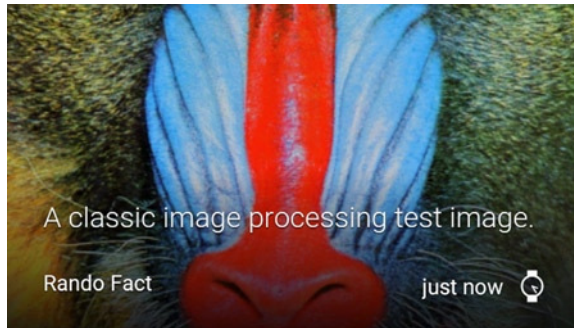


Figure 3-6. A `BigPictureStyle` notification shared between a handheld and Glass. On Glass, the notification displays the picture as the background of a card that displays additional information as a caption

InboxStyle Notifications

In Android Wear, an `InboxStyle` notification appears as a card in the context stream (see Figure 3-7). If the notification has too many items to fit in a single card, the card can be expanded by tapping on it.

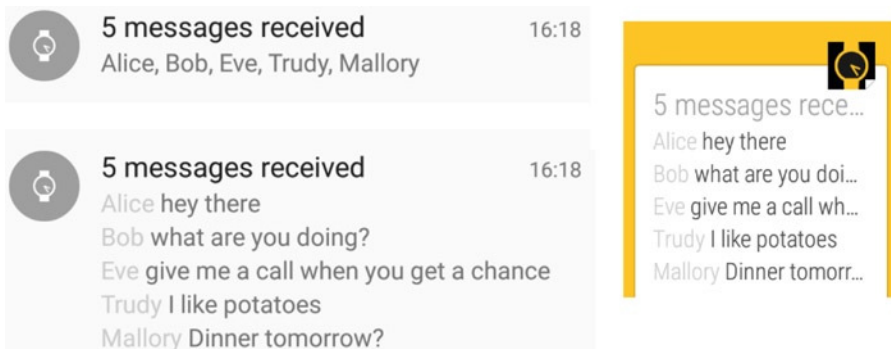


Figure 3-7. An `InboxStyle` notification shared between a handheld and Android Wear. The unexpanded view on the handheld (top left). The expanded view on the handheld (bottom left). In Android Wear, the notification’s content appears in a card in the context stream (right)

In Glass, an `InboxStyle` notification appears as a static card in the past section of the timeline. If the notification has too much information to fit in a single card, users can view the entire content by tapping on the “Show more” menu item. (See Figure 3-8).

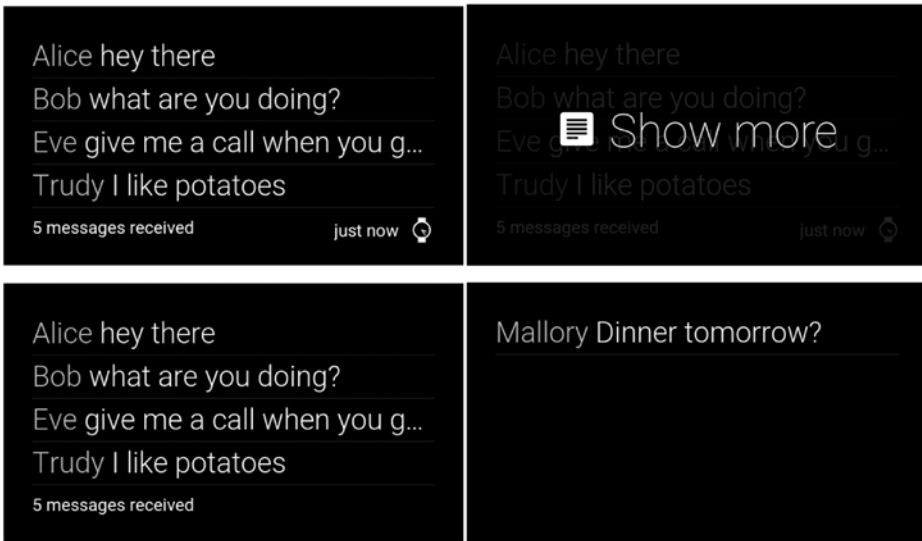


Figure 3-8. An InboxStyle notification shared between a handheld and Glass. The notification is displayed as a static card, and tapping on the “Show more” menu item reveals additional content (top). Additional content is displayed as a scrollable list of cards (bottom)

Notification Actions

In Android Wear, every action of a notification appears as a full screen button. Tapping a button displays a confirmation animation on the watch and triggers the appropriate PendingIntent on the handheld (see Figure 3-9).

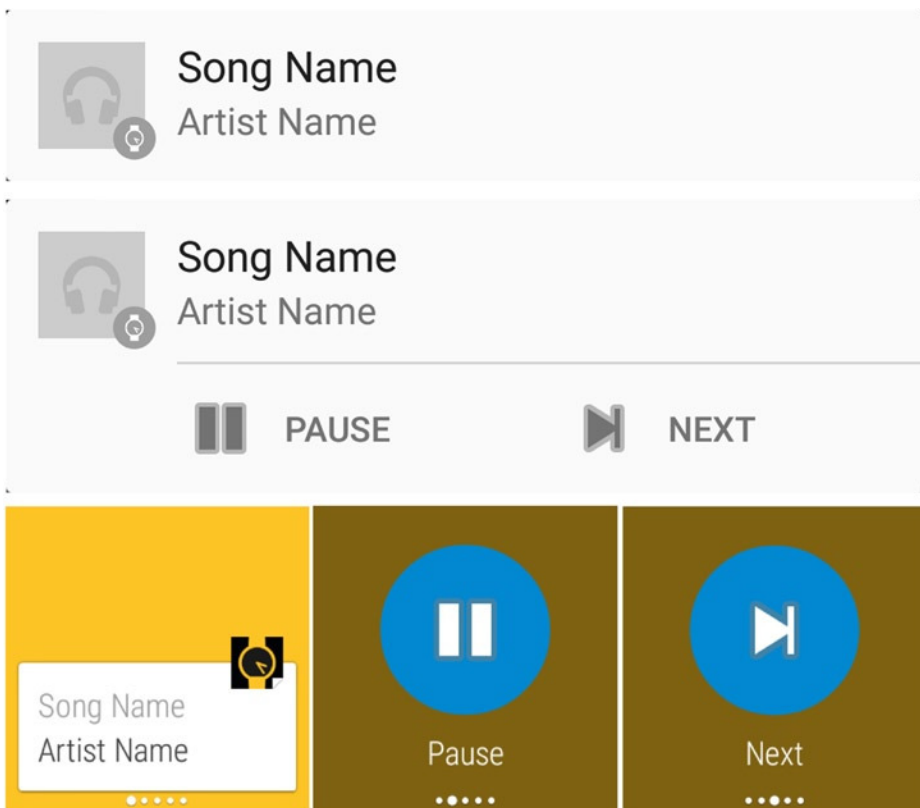


Figure 3-9. A notification with actions shared between a handheld and Android Wear. The unexpanded view in the handheld (top). The expanded view in the handheld (middle). In Android Wear, the notification displays action buttons on additional pages. Tapping on an action triggers the action's respective intent on the handheld (bottom)

In Glass, notification actions appear as menu items. Selecting a menu item for a notification action triggers the appropriate `PendingIntent` on the handheld (see Figure 3-10).

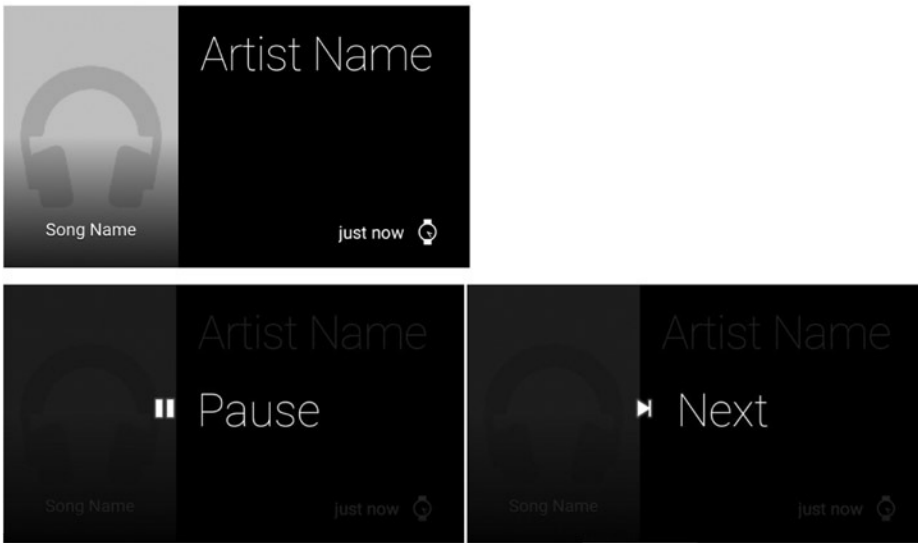


Figure 3-10. A notification with actions shared between a handheld and Android Glass. A static card displays the basic content of the notification (top). Notification actions appear as menu items, and selecting an item triggers the appropriate `PendingIntent` on the handheld (bottom)

Building these notifications is simple and convenient to implement because they require no wearable-specific code. However, customizing notifications for wearables can increase their effectiveness, as we'll see throughout the rest of the chapter.

Customizing Notifications for Wearables

With handheld notifications, the main content of the app is just a tap away. For instance, if you receive chat messages and you want to view the entire conversation, all you have to do is to tap on the notification. On the other hand, tapping on a wearable notification opens the content on the handheld and not directly on the wearable. Alas, tapping on a wearable notification only to take out your phone is not convenient and should be avoided if the additional content is simple enough to be viewed directly on a wearable. In this section, we'll improve wearable notifications by adding wearable-specific code.

Wielding `TaskStackBuilder`

A task is a stack of activities. In handhelds, when you select an app from the launcher and navigate from activity to activity, activities are added to the task stack as they leave the foreground. Pressing the back button causes the current activity to leave the foreground, after which the activity at the top of the task stack enters the foreground.

If you are using a chat app, launching the handheld app starts an activity that displays your friend list. When you press a friend's name, the friend list activity is stopped and added to the task stack and the chat activity is started. Then, if you press the back button, the chat activity is stopped and the friend list activity is removed from the task stack and placed in the foreground.

Consider this alternative scenario: There are no activities from the chat app in the foreground or in the task stack. A friend sends you a message and you receive a notification. Tapping the notification opens the chat activity directly (that is, without going through the friend list activity). You then press the back button. What happens next? It depends on the `PendingIntent` used as the notification's content intent. If it was obtained with `PendingIntent.getActivity`, then the chat activity gets stopped and the launcher enters the foreground.

The Android design guidelines state that this behavior is undesirable (see <http://developer.android.com/design/patterns/navigation.html#into-your-app>). When you press the back button on the chat activity, it should take you to the friend list activity. To achieve this behavior, use `TaskStackBuilder` to manually build the desired task stack when you create the notification's `PendingIntent`. The notification in Figure 3-11 has a content intent that was built with `TaskStackBuilder`, and we'll see how to implement it shortly.

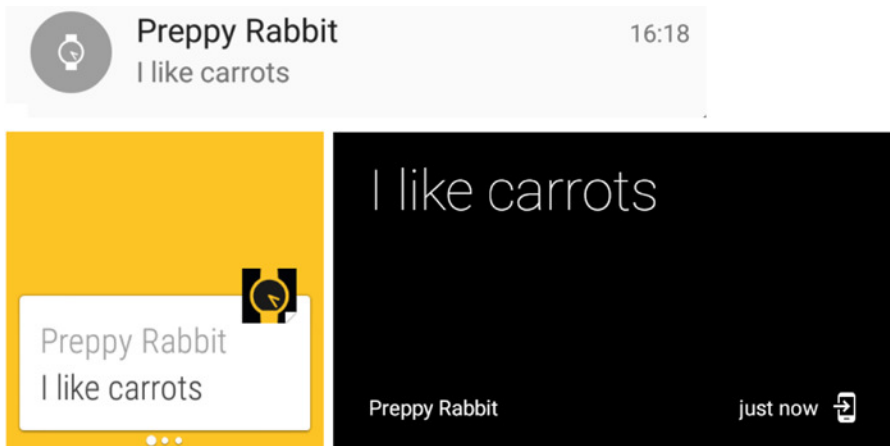


Figure 3-11. A notification with a content intent created with `TaskStackBuilder` as shown on Android Wear (left) and Glass (right)

Note A more comprehensive discussion of tasks and the task stack is outside the scope of this book. If you are not comfortable with these concepts, you can find detailed explanations in the official documentation (<http://developer.android.com/guide/components/tasks-and-back-stack.html>) and on the *Google IO 2012* session on *Navigation in Android* (<https://www.youtube.com/watch?v=XwGHJJYBs0Q>).

Implementation

The content intent of the notification from Figure 3-11 opens `ChatDetailActivity` with the correct task stack so that pressing the back button always starts `MainActivity`. This section demonstrates how to implement this notification.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the first button (which is labeled *Build Task Stack*) to trigger the notification.

Implement `ChatDetailActivity`

Create a new activity called `ChatDetailActivity`, which is a placeholder for the history of a conversation with a particular user. In a real chat app, it would contain the conversation history and it would let you send messages to the user. However, since our focus is on notifications, we won't place any content in this activity, other than a short placeholder message.

1. Create a layout.

In res ► layout ► `activity_chat_detail.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_margin="16dip"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:text="Conversation History Here"
        android:textStyle="bold"
        android:textAppearance="@android:style/TextAppearance.Large"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:text="This Activity is a placeholder for a conversation's history. The app
        only demonstrates how to implement notifications and not an entire chat app."
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/reply"
        android:layout_marginTop="32dip"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

2. Declare `ChatDetailActivity` and list `MainActivity` as its logical parent.

In `AndroidManifest.xml`:

```
<activity
    android:name=".ChatDetailActivity"
    android:label="Conversation"
    android:parentActivityName=".MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```

The key part of this step is `android:parentActivityName=".MainActivity"`, which specifies that `MainActivity` is the logical parent of `ChatDetailActivity`. In the next section, `TaskStackBuilder` will use this information to construct the proper task stack. The meta-data tag in this activity declaration replaces `parentActivityName` in Android versions prior to API level 16. Since our `minSdkVersion` is 16, this meta-data tag is not used and only serves a demonstration purpose.

3. Implement `ChatDetailActivity`. It should set the action bar's title to the name of the user one is chatting with.

In `ChatDetailActivity.java`:

```
public class ChatDetailActivity extends ActionBarActivity {
    public static final String EXTRA_CHATTING_WITH = "chatting_with";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_chat_detail);

        String chattingWith = getIntent().getStringExtra(EXTRA_CHATTING_WITH);
        if(chattingWith != null) {
            getSupportActionBar().setTitle(chattingWith);
        }
    }
}
```

Note that `ChatDetailActivity` lets you set the username of the person you're chatting with by using an Intent extra.

Implement the Notification

The class we just implemented, `ChatDetailActivity`, is a placeholder for a conversation in a chat message. Tapping on the chat notification should take users to this activity.

1. Create a helper method that returns a `PendingIntent` that starts the `ChatDetailActivity` with the correct task stack.

The first parameter of `getConversationPendingIntent` (that is, `chattingWith`) is a `String` that contains the username of the person you're chatting with. This `String` will be given to `ChatDetailActivity` as an extra so it can be displayed as the action bar title.

In `MainActivity.java`:

```
private PendingIntent getConversationPendingIntent(String chattingWith, int requestCode) {
    Intent conversationIntent = new Intent(this, ChatDetailActivity.class);

    if(chattingWith != null) {
        conversationIntent.putExtra(ChatDetailActivity.EXTRA_CHATTING_WITH, chattingWith);
    }

    TaskStackBuilder taskStackBuilder = TaskStackBuilder.create(this);
    taskStackBuilder.addParentStack(ChatDetailActivity.class);
    taskStackBuilder.addNextIntent(conversationIntent);

    return taskStackBuilder.getPendingIntent(requestCode, PendingIntent.FLAG_CANCEL_CURRENT);
}
```

If the user closes the app and presses the notification, it will start `ChatDetailActivity`. If the user then presses the back button, `ChatDetailActivity` stops and reveals `MainActivity` underneath. If the `PendingIntent` used to start `ChatDetailActivity` simply started the `ChatDetailActivity`, then pressing the back button would take the user back to the launcher screen. To ensure proper navigation, `TaskStackBuilder` creates a `PendingIntent` that takes into account the logical parent of `ChatDetailActivity`.

The `addParentStack` method looks at the meta-data of the specified activity to find its logical parent (that is, it looks at `android:parentActivityName=".MainActivity"`). Note that the `addParentStack` method only adds the logical parents (and grandparents, if available) of the given activity to the task stack; it does not add the activity itself. The `addNextIntentMethod` adds the activity itself and finishes building the proper task stack.

Place the code from steps 2 and 3 where you want to create the notification. In the sample project, we call this code in the `onBuildTaskStackContentIntentClick` method.

2. Create the notification and specify the `PendingIntent` from step 2 as its content intent.

```
PendingIntent conversationPendingIntent = getConversationPendingIntent("Preppy Rabbit", 0);

Notification notification = new NotificationCompat.Builder(this)
    .setContentTitle("Preppy Rabbit")
    .setContentText("I like carrots")
    .setSmallIcon(R.drawable.ic_stat_notify)
```

```

.setContentIntent(conversationPendingIntent)
.setCategory(Notification.CATEGORY_MESSAGE)
.setPriority(Notification.PRIORITY_HIGH)
.setDefaults(Notification.DEFAULT_ALL)
.build();

```

3. Issue the notification.

```

NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, notification);

```

This example does not utilize any wearable-specific functionality but provides a foundation for the rest of the examples in this chapter.

Wearable-Only Actions

By default, every notification action appears on both the handheld and the wearable. Thus, the same `PendingIntent` is triggered regardless of whether you tap on the action on the handheld or on the wearable. While this behavior is desired most of the time, there are situations in which you need to have different actions on the handheld than on the wearable. For instance, if you are collecting analytics (that is, data that shows how people are using your app), you could record whether a notification action was triggered from a handheld or wearable by using two different actions.

The following implementation creates actions that appear only on the handheld, only on the wearable, or on both devices (see Figure 3-12).

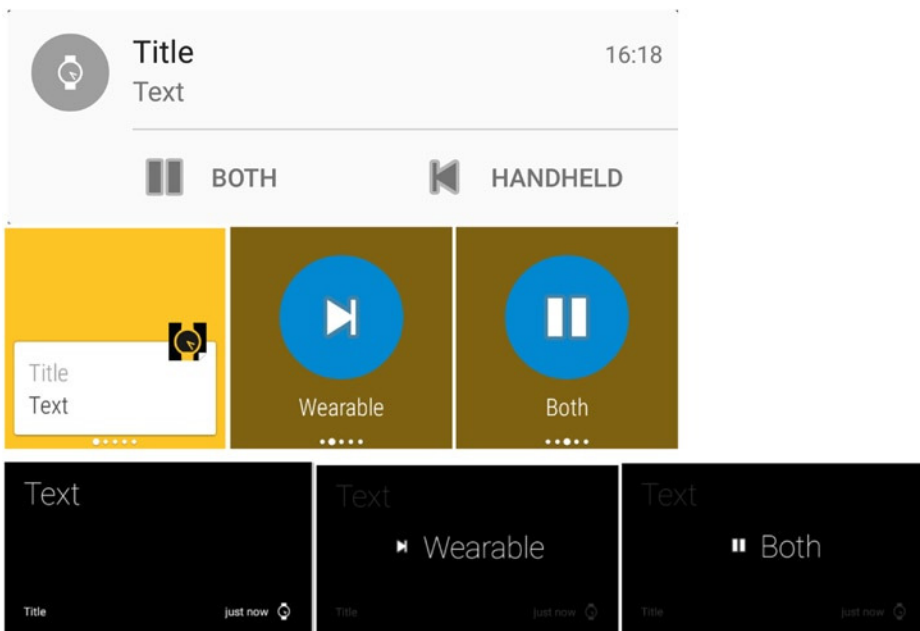


Figure 3-12. A notification displays actions on a handheld device (top), Android Wear device (middle), and Glass (bottom). The “Handheld” action is unique to the handheld notification, the “Wearable” action is unique to the wearable, and the “Both” actions appear on all devices

Implementation

This section implements a notification with an action that only appears on the wearable, an action that appears only on the handheld, and an action that appears on both devices.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the second button (which is labeled *Wearable Only Actions*) to trigger the notification.

Implement ActionFeedbackActivity

This activity provides feedback when an action is triggered and is used only for demonstration purposes.

1. Create a layout.

In res ► layout ► activity_action_feedback.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/action_feedback"
    android:layout_margin="16dip"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

2. Declare the activity.

In AndroidManifest.xml:

The `android:parentActivityName`, as we discussed in the previous example, specifies that `MainActivity` is the logical parent of `ActionFeedbackActivity`.

```
<activity
    android:name=".ActionFeedbackActivity"
    android:label="Action Feedback"
    android:launchMode="singleTop"
    android:parentActivityName=".MainActivity" >
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity" />
</activity>
```

3. Implement the activity.

In `ActionFeedbackActivity.java`:

```
public class ActionFeedbackActivity extends ActionBarActivity {
    public static final String EXTRA_ACTION_FEEDBACK = "action_feedback";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_action_feedback);

        String actionFeedback = getIntent().getStringExtra(EXTRA_ACTION_FEEDBACK);
        if(actionFeedback != null) {
            ((TextView) findViewById(R.id.action_feedback)).setText(actionFeedback);
        }
    }
}
```

Implement the Notification

Start by creating a few helper methods.

1. Create a helper method that returns a `PendingIntent` that starts `ActionFeedbackActivity` with the proper task stack.

The notification uses this method to create `PendingIntents` for its actions.

In `MainActivity.java`:

```
private PendingIntent getActionFeedbackPendingIntent(String actionFeedback, int requestCode)
{
    Intent actionFeedbackIntent = new Intent(this, ActionFeedbackActivity.class);
    actionFeedbackIntent.putExtra(ActionFeedbackActivity.EXTRA_ACTION_FEEDBACK, actionFeedback);

    TaskStackBuilder taskStackBuilder = TaskStackBuilder.create(this)
        .addParentStack(ActionFeedbackActivity.class)
        .addNextIntent(actionFeedbackIntent);

    return taskStackBuilder.getPendingIntent(requestCode, PendingIntent.FLAG_CANCEL_CURRENT);
}
```

2. Create a helper method that returns a `PendingIntent` that starts `MainActivity`.

The notification uses this `PendingIntent` as its content intent. That is, tapping on the notification opens `MainActivity`.

In `MainActivity.java`:

```
private PendingIntent getMainActivityPendingIntent() {
    Intent mainActivityIntent = new Intent(this, MainActivity.class);
    mainActivityIntent.addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);
    return PendingIntent.getActivity(this, 0,
        mainActivityIntent, PendingIntent.FLAG_CANCEL_CURRENT);
}
```

We will reuse this helper method in most of the subsequent examples in this chapter. The flag `Intent.FLAG_ACTIVITY_SINGLE_TOP` ensures that an `Activity` does not get started multiple times even if it is already at the top of the task stack (that is, in the foreground).

Place the code from steps 3–5 where you want to create the notification. In the sample project, we call this code in the `onWearableOnlyActionsClick` method.

3. Create the three notification Actions.

```
PendingIntent handheldActionFeedbackPendingIntent =
    getActionFeedbackPendingIntent("You invoked the handheld only action", 0);

PendingIntent wearableActionFeedbackPendingIntent =
    getActionFeedbackPendingIntent("You invoked the wearable only action", 1);

PendingIntent bothActionFeedbackPendingIntent =
    getActionFeedbackPendingIntent("You invoked the action that appears on both devices", 2);

NotificationCompat.Action handheldOnlyAction = new NotificationCompat.Action(
    android.R.drawable.ic_media_previous, "Handheld",
    handheldActionFeedbackPendingIntent);
NotificationCompat.Action wearableOnlyAction = new NotificationCompat.Action(
    android.R.drawable.ic_media_next, "Wearable", wearableActionFeedbackPendingIntent);
NotificationCompat.Action action = new NotificationCompat.Action(
    android.R.drawable.ic_media_pause, "Both", bothActionFeedbackPendingIntent);
```

The `NotificationCompat.Action` constructor takes three parameters:

- a drawable resource for the icon that represents the action,
- the title of the action, and
- a `PendingIntent` that will be triggered when the action is executed.

For the purposes of this sample, we're using arbitrary icons.

4. Create a Notification with the Actions obtained from the previous step.

```
NotificationCompat.WearableExtender wearableExtender = new NotificationCompat.
WearableExtender()
    .addAction(wearableOnlyAction)
    .addAction(action);

PendingIntent mainActivityPendingIntent = getMainActivityPendingIntent();

Notification notification = new NotificationCompat.Builder(this)
    .setContentTitle("Title")
    .setContentText("Text")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setContentIntent(mainActivityPendingIntent)
    .setCategory(Notification.CATEGORY_STATUS)
    .addAction(action)
    .extend(wearableExtender)
    .addAction(handheldOnlyAction)
    .build();
```

`WearableExtender` is the most important class in this entire chapter. It lets you extend handheld notifications with wearable-specific functionality. In this case, the actions you add to the instance of `WearableExtender` will appear exclusively on the wearable. Furthermore, all other actions (that is, those added directly to the notification) will appear exclusively on the handheld. As a result, any action that you want to appear on both devices must be added to both the `WearableExtender` and the `Notification`. If, on the other hand, you never add any actions to the `WearableExtender`, then all actions that you add to the notification appear on both devices.

Once you have created a `WearableExtender`, associate it with a `Notification` by giving it as a parameter to `NotificationCompat.Builder`'s `extend` method.

5. Issue the Notification.

```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, notification);
```

Subsequent examples will expand on the usage of `WearableExtender`.

Notification Pages

Since wearable devices have a different form factor from handhelds, they require distinct user interfaces. Many notifications that are appropriate for handhelds are not effective on wearables. Consider a notification that you receive when a friend sends you a chat message. The handheld notification displays your friend's username, avatar, and message. Viewing the recent conversation history is as easy as tapping the notification. The conversation provides context and reminds you of what you were talking about.

With wearable notifications, on the other hand, it becomes more tedious to view the recent conversation history. If you tap the content intent (that is, the "Open on phone" button or menu item), the conversation pops up on the phone. While wearables should redirect users to their handhelds for long or complex interactions, having to take out your phone for simple tasks should be avoided. Instead, we can use notification paging to show the recent conversation history directly on the wearable.

Notification pages are additional screens of content that provide additional content on a wearable. In Android Wear, pages appear immediately to the right of a main notification card where a user can swipe back and forth from right to left to navigate between pages. In Glass, pages appear as a scrollable list of cards that can be accessed by tapping on the "Show more" menu item. In the case of the chat notification, we can use a page to display the recent conversation history as shown in Figure 3-13 for Android Wear and Figure 3-14 for Glass.

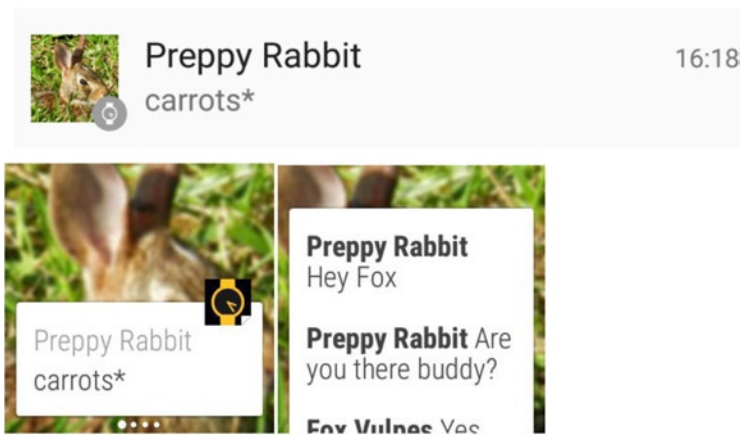


Figure 3-13. A notification indicates that a new chat message was received. Unlike the handheld notification (top), the Android Wear notification (bottom) shows the recent conversation history in addition to the latest message

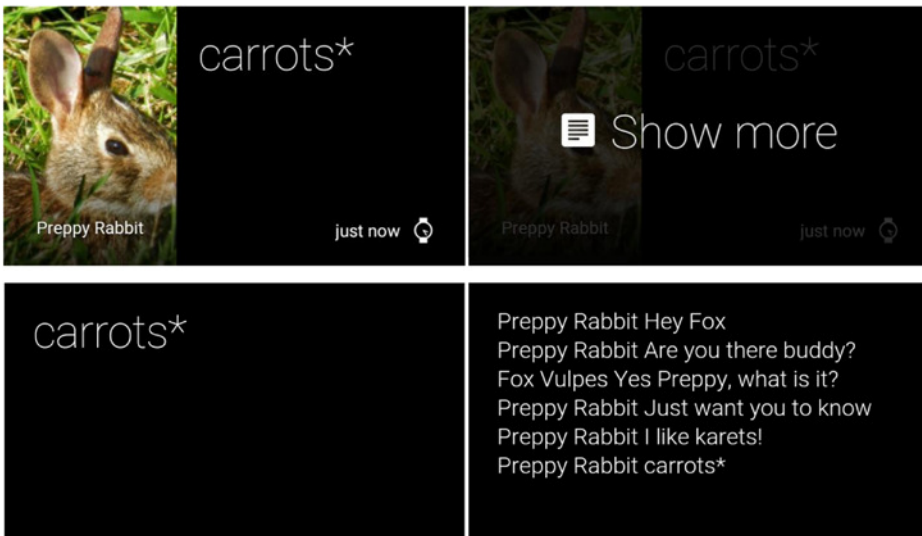


Figure 3-14. A notification with pages on Glass. To access additional pages, users select the “show more” menu item (top). Additional pages appear as scrollable cards (bottom)

While the main page of the wearable notification only contains the latest chat message received, the second page displays the recent conversation history, which gives the user context without having to take out his handheld. Thanks to notification paging, the user does not need to take out his phone to understand the context of the latest message.

BigTextStyle Pages on Android Wear and Glass

In Android Wear, a `BigTextStyle` page does not require a content title or content text because the big text determines the entire content of the card. Tapping on the card expands the card into a scrollable card that displays the entire content of the text.

With Glass (as of XE22), in contrast, a `BigTextStyle` page completely ignores the big text. Instead, pages on Glass require a content text. Unlike with Android Wear, the content text in Glass can occupy multiple pages.

To implement a `BigTextStyle` page that works with both Android Wear and Glass, set both the big text and the content text to the same long message.

Implementation

This section implements a notification with pages that plays a sound and vibrates when issued.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the third button (which is labeled *Paging*) to trigger the notification. For this sample, make sure that your handheld's notification volume is not muted or the notification will not play an alert or vibrate. If you have a Nexus 7, note that the device does not support vibration.

1. Create a `CharSequence` that contains the conversation history you want to display. Here, we'll hard-code a random sample message.

In `MainActivity.java`:

```
private CharSequence generateSampleMessage1() {
    return TextUtils.concat(formatMessage("Preppy Rabbit", "Hey Fox"), "\n\n",
        formatMessage("Preppy Rabbit", "Are you there buddy?"), "\n\n",
        formatMessage("Fox Vulpes", "Yes Preppy, what is it?"), "\n\n",
        formatMessage("Preppy Rabbit", "Just want you to know"), "\n\n",
        formatMessage("Preppy Rabbit", "I like karets!"), "\n\n",
        formatMessage("Preppy Rabbit", "carrots*"));
}

private Spannable formatMessage(String author, String message) {
    Spannable spannable = new SpannableString(author + " " + message);
    spannable.setSpan(new StyleSpan(Typeface.BOLD), 0, author.length(),
        Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
    return spannable;
}
```

The recent conversation history page in the wearable notification (see Figure 3-14) is nothing more than a `BigTextStyle` notification. The usernames are bolded to separate them from the rest of the text, and adjacent messages are separated by two new lines. The formatting appears as expected on Android Wear, but Glass ignores it (see Figure 3-14). The `formatMessage` method concatenates a username and a message and uses a `Span` to bold the username.

2. Create a helper method that scales a Bitmap to the dimensions required by a notification's large icon. The large icon is used to display the user's avatar.

In MainActivity.java:

```
private Bitmap getScaledLargeIconFromResource(int resource) {
    Resources res = getResources();
    int height = (int) res.getDimension(android.R.dimen.notification_large_icon_height);
    int width = (int) res.getDimension(android.R.dimen.notification_large_icon_width);
    Bitmap largeIcon = BitmapFactory.decodeResource(res, resource);
    return Bitmap.createScaledBitmap(largeIcon, width, height, false);
}
```

The dimensions of the Bitmap are usually 64x64 dips, but we obtain these values from Android's system resources using the `getScaledLargeIconFromResource` method because hard-coding values is not good style. The method `Bitmap.createScaledBitmap` scales another Bitmap given as its first parameter to the dimensions specified by its next two parameters.

Place the code from steps 3–5 where you want to create the notification. In the sample project, this code resides in the `onPagingClick` method.

3. Build a `BigTextStyle` notification for the additional page. As discussed above, the `Notification` should set both the content text and the big text to the recent conversation history.

```
CharSequence message = generateSampleMessage1();

NotificationCompat.BigTextStyle bigTextStyle = new NotificationCompat.BigTextStyle()
    .bigText(message);

Notification messageHistoryPage = new NotificationCompat.Builder(this)
    .setContentText(message) // needed for Glass
    .setStyle(bigTextStyle)
    .build();
```

4. Create the Notification.

```
NotificationCompat.WearableExtender wearableExtender = new NotificationCompat.
WearableExtender()
    .addPage(messageHistoryPage);

Bitmap preppyAvatar = getScaledLargeIconFromResource(R.drawable.preppy);

Notification notificationWithPages = new NotificationCompat.Builder(this)
    .setContentTitle("Preppy Rabbit")
    .setContentText("carrots*")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setContentIntent(getConversationPendingIntent("Preppy Rabbit", 0))
    .setCategory(Notification.CATEGORY_MESSAGE)
```

```

.setLargeIcon(preppyAvatar)
.setPriority(Notification.PRIORITY_HIGH)
.setDefaults(Notification.DEFAULT_ALL)
.extend(wearableExtender)
.build();

```

5. The user's avatar is displayed as the notification's large icon, which appears on both the handheld and wearable.

Finally, trigger the Notification.

```

NotificationManagerCompat notificationManager =
    NotificationManagerCompat.from(MainActivity.this);
notificationManager.notify(NOTIFICATION_ID, notificationWithPages);

```

Keep in mind that adding notification pages does not affect the handheld notification in any way and that their purpose is only to enhance the wearable user experience.

Stacking Notifications

What happens if you receive chat messages from multiple users? The handheld notification can preview the latest messages received from all users with `InboxStyle`, as shown at the top of Figure 3-15.

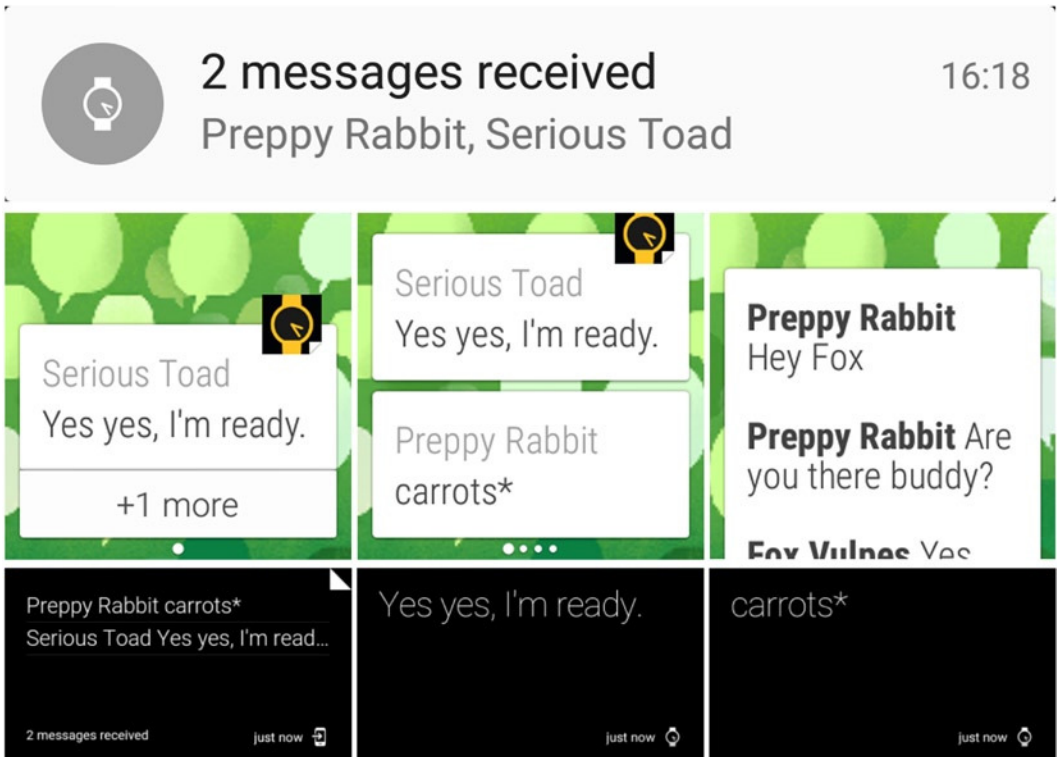


Figure 3-15. A notification issued when multiple chat messages are received. The handheld notification shows an `InboxStyle` preview of each message received (top), the Android Wear notification contains the recent conversation history as a page (middle), and the Glass notification displays all messages in a bundle (bottom)

However, by themselves, `InboxStyle` notifications are not effective on wearables because they can only display a short preview of each message. This preview may not necessarily contain the complete message, and users have no way of looking at the recent conversation history other than looking at the handheld. Users would essentially be getting notifications for incomplete messages, which is far from ideal. We don't want to force users to take out their handhelds every time they receive multiple chat messages.

Could we simply create multiple notifications, one for each message? We could, but that's a really bad idea. We don't want to make the user scroll through the context stream in search of additional messages that may or may not be there. If only there were a way to aggregate multiple notifications onto a single page...

Enter stacking notifications. Stacks group multiple notifications and associate them by placing them on the same page, as shown at the middle and bottom of Figure 3-15.

In Android Wear, the primary page of a notification stack contains multiple cards. Users can access additional pages by selecting one of these cards and swiping from right to left. In Glass, a bundle of cards contains all the notifications that are part of the stack.

Implementation

This section implements a stack of notifications.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the fourth button (which is labeled *Stacking*) to trigger a stack of notifications.

1. Generate a sample conversation history. Note that the `formatMessage` method is part of the previous example on paging.

In `MainActivity.java`:

```
private CharSequence generateSampleMessage2() {
    return TextUtils.concat(formatMessage("Fox Vulpes", "We're heading out. Are you ready?"), "\n\n",
        formatMessage("Serious Toad", "I think so!"), "\n\n",
        formatMessage("Serious Toad", "Yes yes, I'm ready."));
}
```

2. Create a style for a span that highlights the username of a notification.

In `res > values > styles.xml`:

```
<style name="NotificationPrimaryText">
    <item name="android:textColor">#cccccc</item>
</style>
```

3. Create a helper method to format an `InboxStyle` line to preview a username and message. This method uses a `Span` based on the style defined in step 2 to lighten the color of the username and make it easy to distinguish from the message.

In `MainActivity.java`:

```
private Spannable formatInboxStyleLine(String username, String message) {
    TextAppearanceSpan notificationSenderSpan = new TextAppearanceSpan(this,
        R.style.NotificationPrimaryText);
    Spannable spannable = new SpannableString(username + " " + message);
    spannable.setSpan(notificationSenderSpan, 0, username.length(),
        Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
    return spannable;
}
```

In the previous chapter, we used a `ForegroundColorSpan` that creates the same appearance for `InboxStyle` lines, but we're using a `TextAppearanceSpan` in this chapter to show you an alternative way of achieving the same effect.

4. Create a helper method to create a notification for an individual message. This notification is very similar to the one we wrote in the paging sample.

Every notification that's part of a stack should have the same group identifier, which is a unique `String`.

```
private Notification generateSampleNotification(String author, String message,
    CharSequence messageHistory, int requestCode) {
    NotificationCompat.BigTextStyle bigTextStyle = new NotificationCompat.BigTextStyle()
        .bigText(messageHistory);

    Notification messageHistoryPage = new NotificationCompat.Builder(this)
        .setStyle(bigTextStyle)
        .build();

    NotificationCompat.WearableExtender wearableExtender = new NotificationCompat.
        WearableExtender()
        .addPage(messageHistoryPage);

    Notification notificationWithPages = new NotificationCompat.Builder(this)
        .setTitle(author)
        .setContent(message)
        .setSmallIcon(R.drawable.ic_launcher)
        .setContentIntent(getConversationPendingIntent(author, requestCode))
        .setGroup(GROUP_KEY_MESSAGES)
        .extend(wearableExtender)
        .build();

    return notificationWithPages;
}
```

Where `GROUP_KEY_MESSAGES` is a `String` that uniquely identifies a stack of notifications.

Place the code from steps 5–8 where you want to issue the notification. In the sample project, this code resides in the `onStackingClick` method in `MainActivity.java`.

5. For every notification, fetch the usernames (the authors), latest messages, and recent conversation histories. Note that the `generateSampleMessage1` method is part of the previous sample on paging.

```
String[] authors = { "Preppy Rabbit", "Serious Toad" };
String[] messages = { "carrots*", "Yes yes, I'm ready." };
CharSequence[] messageHistories = { generateSampleMessage1(), generateSampleMessage2() };
```

6. Create a summary notification that will be displayed only on the handheld. Although this notification will not appear on the wearable, the background given to its `WearableExtender` will be used as the background of the wearable notification. Note that the `getScaledLargeIconFromResource` method is part of the previous sample on paging, while the `getMainActivityPendingIntent` method is part of the wearable only actions sample.

In addition to setting the same group identifier as the rest of the items in the notification stack, the summary notification should call `setGroupSummary(true)`.

```
NotificationCompat.WearableExtender wearableExtender = new NotificationCompat.WearableExtender()
    .setBackground(getScaledLargeIconFromResource(R.drawable.wear_bg));
```

```
NotificationCompat.InboxStyle inboxStyle = new NotificationCompat.InboxStyle()
    .addLine(formatInboxStyleLine(authors[0], messages[0]))
    .addLine(formatInboxStyleLine(authors[1], messages[1]));
```

```
PendingIntent mainActivityPendingIntent = getMainActivityPendingIntent();
```

```
Notification summaryNotification = new NotificationCompat.Builder(this)
    .setContentTitle("2 messages received")
    .setContentText("Preppy Rabbit, Serious Toad")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setContentIntent(mainActivityPendingIntent)
    .setCategory(Notification.CATEGORY_MESSAGE)
    .setPriority(Notification.PRIORITY_HIGH)
    .setDefaults(Notification.DEFAULT_ALL)
    .extend(wearableExtender)
    .setGroup(GROUP_KEY_MESSAGES)
    .setGroupSummary(true)
    .setStyle(inboxStyle)
    .build();
```

Each notification in the stack still needs the three required elements of every notification: the content title, content text, and the small icon. If any of these attributes is missing, your entire notification stack may not work as expected.

7. Create the sample notifications.

```
Notification messageNotification1 = generateSampleNotification(authors[0], messages[0],
    messageHistories[0], 0);
Notification messageNotification2 = generateSampleNotification(authors[1], messages[1],
    messageHistories[1], 1);
```

8. Issue all of the notifications with distinct notification IDs. Even though we're issuing three notifications, the summary notification will only appear on the phone and the notification stack will only appear on the wearable. Android knows all these notifications should be displayed as a stack because they belong to the same group.

```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, messageNotification1);
notificationManager.notify(NOTIFICATION_ID+1, messageNotification2);
notificationManager.notify(NOTIFICATION_ID+2, summaryNotification);
```

In summary, stacking notifications allows users to read the details of every message directly on the wearable without creating multiple notifications.

Voice Input Notification

With voice input, users can reply to a chat message directly on a wearable by selecting a predefined text option or dictating a reply with voice recognition. Once users have finished replying, the handheld app receives the response as an extra in an activity or service.

Figure 3-16 and Figure 3-17 depict how users reply directly on Android Wear and Glass, respectively. When users select the “Reply” action, a wearable requests voice input. In Android Wear, users can either swipe up and down to navigate predefined responses (if there are any available) or speak their own reply. In Glass, users can only speak their reply.

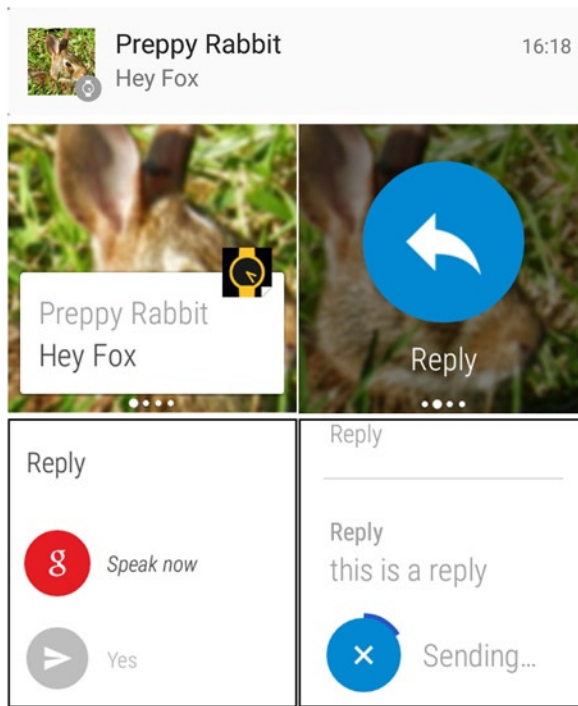


Figure 3-16. A notification for a received chat message. Users may tap the handheld notification to open the app and type a reply (top). In Android Wear, a notification action allows users to reply directly from the watch (middle). When users reply, they can select a predefined response or speak a reply (bottom)

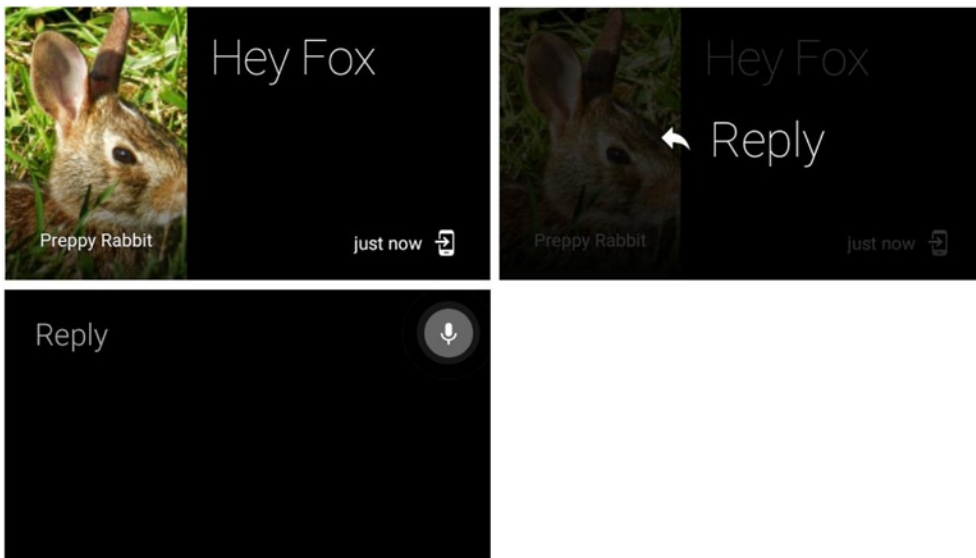


Figure 3-17. A notification for a received chat message. On Glass, a menu item allows users to reply directly from the device (top). Users reply using voice recognition and, unlike with Android Wear, are not able to choose from predefined responses (bottom)

Implementation

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the fifth button (which is labeled *Voice Input*) to trigger a notification that requests voice input.

This section implements a notification that requests voice input.

Update ChatDetailActivity

Modify `ChatDetailActivity` to also display a `TextView` that displays the voice reply for feedback.

1. Update the layout.

Add the following `TextView` to `res > layout > activity_action_feedback.xml`, right before the closing `LinearLayout` tag:

```
<TextView
    android:id="@+id/reply"
    android:layout_marginTop="32dip"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

2. Update `ChatDetailActivity.java`.

```
public class ChatDetailActivity extends ActionBarActivity {
    public static final String EXTRA_VOICE_REPLY = "extra_voice_reply";
    public static final String EXTRA_CHATTING_WITH = "chatting_with";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_chat_detail);

        CharSequence replyText = getMessageText(getIntent());
        if(replyText != null) {
            TextView replyTextView = (TextView)findViewById(R.id.reply);
            replyTextView.setText("You replied: " + replyText);
        }

        String chattingWith = getIntent().getStringExtra(EXTRA_CHATTING_WITH);
        if(chattingWith != null) {
            getSupportActionBar().setTitle(chattingWith);
        }
    }
}
```

```

private CharSequence getMessageText(Intent intent) {
    Bundle remoteInput = RemoteInput.getResultsFromIntent(intent);
    if (remoteInput != null) {
        return remoteInput.getCharSequence(EXTRA_VOICE_REPLY);
    }
    return null;
}
}

```

The `getMessageText` method shows you how to extract the voice reply from the `Intent`.

Implement the Notification

Place the code from steps 1–4 where you want to issue the notification. In the sample project, we do so in the `onVoiceReplyClick` method.

1. Create an array of predefined response and use it to build an instance of `RemoteInput`.

```

String[] choices = new String[] { "Yes", "No", "In a meeting" };

RemoteInput remoteInput = new RemoteInput.Builder(ChatDetailActivity.EXTRA_VOICE_REPLY)
    .setLabel("Reply")
    .setChoices(choices)
    .setAllowFreeFormInput(false)
    .build();

```

`RemoteInput` is a class that provides Android Wear with the information needed to create a voice input screen. In particular, `RemoteInput` specifies the label at the top of the voice input screen as well as a list of predefined responses. If the notification should accept speech replies that may be different from the predefined responses, call `setAllowFreeFormInput(true)`.

Note Glass does not accept predefined responses and will accept freeform input even if the option is disabled. Only Android Wear supports predefined responses.

2. Create a notification `Action` and associate it with the instance of `RemoteInput` from the previous step.

```

PendingIntent replyPendingIntent = getConversationPendingIntent("Preppy Rabbit", 0);

NotificationCompat.Action replyAction =
    new NotificationCompat.Action.Builder(R.drawable.ic_full_reply, "Reply", replyPendingIntent)
        .addRemoteInput(remoteInput)
        .build();

```

3. Create a Notification with the action from the previous step. Note that the `getScaledLargeIconFromResource` method is covered in “Notification Pages.”

```
NotificationCompat.WearableExtender wearableExtender = new NotificationCompat.WearableExtender()
    .addAction(replyAction);
```

```
Bitmap preppyAvatar = getScaledLargeIconFromResource(R.drawable.preppy);
```

```
Notification notification = new NotificationCompat.Builder(this)
    .setTitle("Preppy Rabbit")
    .setText("Hey Fox")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setContentIntent(getConversationPendingIntent("Preppy Rabbit", 20))
    .setCategory(Notification.CATEGORY_MESSAGE)
    .setPriority(Notification.PRIORITY_HIGH)
    .setDefaults(Notification.DEFAULT_ALL)
    .setLargeIcon(preppyAvatar)
    .extend(wearableExtender)
    .build();
```

4. Issue the notification.

```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, notification);
```

Even if your wearable notification handles voice input, the user should still be able to click on your handheld notification and reply directly from the handheld. Although we don't implement this functionality in this sample app, recall that every notification should be usable and understandable independently.

Background Only Notifications

`BigPictureStyle` notifications show the big picture as a notification page. However, `BigPictureStyle` is not suitable for all notifications, such as slideshows with multiple images. We can instruct notifications to hide everything but their background to show images without any other content in the way.

The notification in Figure 3-18 and Figure 3-19 (for Android Wear and Glass, respectively) contains a slideshow of two images, each of which is on a single page. In this particular notification, the handheld notification has no expanded state and does not display any images. The user would have to tap on the handheld notification to view both images, although we don't implement that functionality because our focus is on the notification itself.

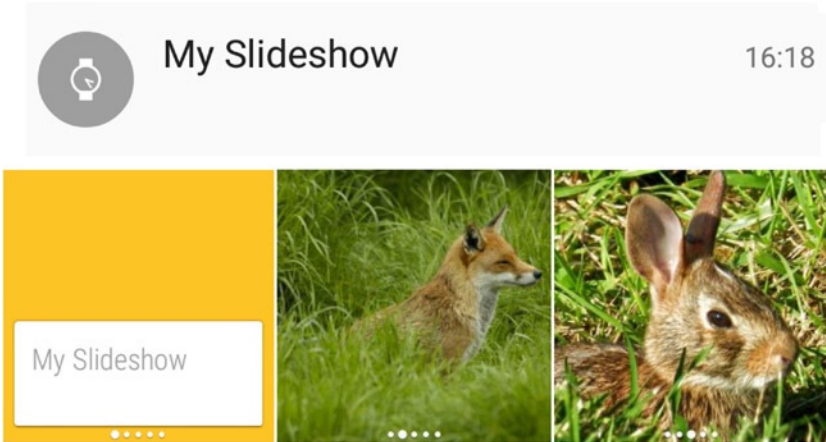


Figure 3-18. Using background only notifications. Tapping on a handheld notification opens an app that would display a slideshow in a real implementation (top). A notification in Android Wear displays both images as the backgrounds of two pages (bottom)

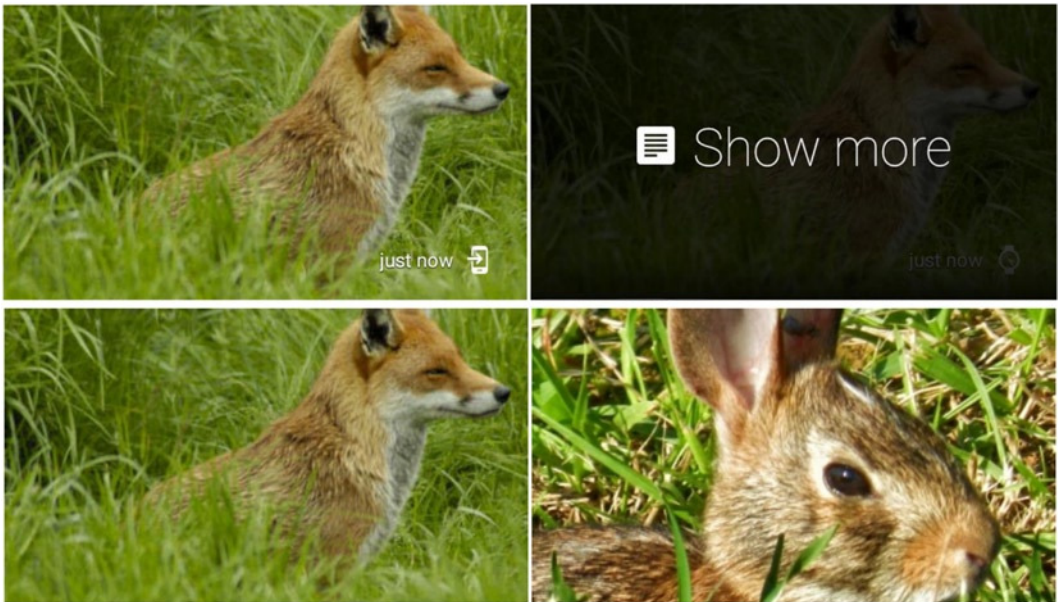


Figure 3-19. Using background only notifications. Selecting the “show more” menu item displays both images on Glass (top). Both images are displayed as the backgrounds of a list of cards (bottom)

Note Place backgrounds for wearable notifications in the `res/drawable-nodpi` folder. For Android Wear, images should have a size of 400x400 or 640x400 if horizontal parallax is desired. Larger images are automatically downsampled. For Glass, images should have a size of 640x360. If your background images target both Android Wear and Glass, a dimension of 640x400 is suitable for both platforms.

Implementation

Let's implement a notification with background-only images.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the sixth button (which is labeled *Background Only Image*) to trigger a notification with background-only images.

1. Create a helper method to return a background only notification. Calling `setHintShowBackgroundOnly(true)` on the `WearableExtender` instance ensures that the content of the notification is hidden.

In `MainActivity.java`:

```
private Notification getImageOnlyNotification(String title, int drawableResource) {
    NotificationCompat.WearableExtender wearableExtender =
        new NotificationCompat.WearableExtender()
            .setBackground(BitmapFactory.decodeResource(getResources(), drawableResource))
            .setHintShowBackgroundOnly(true);

    return new NotificationCompat.Builder(this)
        .setContentTitle(title)
        .setContentText("")
        .setSmallIcon(R.drawable.ic_stat_notify)
        .extend(wearableExtender)
        .build();
}
```

Even though the content of a background notification will be hidden, I still like to set its content title, content text, and small icon to maintain the habit of always setting these fields.

Place the code from steps 2–4 where you want to issue the notification. The sample project calls this code in the `onBackgroundOnlyImageClick` method.

2. Create two background-only notifications.

```
Notification imageOneNotification = getImageOnlyNotification("Fox Avatar", R.drawable.fox);
Notification imageTwoNotification = getImageOnlyNotification("Rabbit Avatar",
    R.drawable.preppy);
```

3. Create a notification and set each of the background-only notifications as pages.

```
NotificationCompat.WearableExtender wearableExtender = new NotificationCompat.
WearableExtender()
    .setHintHideIcon(true)
    .addPage(imageOneNotification)
    .addPage(imageTwoNotification);

Notification notification = new NotificationCompat.Builder(this)
    .setContentTitle("My Slideshow")
    .setContentText("")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setContentIntent(getMainActivityPendingIntent())
    .setCategory(Notification.CATEGORY_SOCIAL)
    .extend(wearableExtender)
    .build();
```

Calling `setHintHideIcon(true)` on the `WearableExtender` hides the icon in the wearable notification.

4. Issue the notification.

```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, notification);
```

`WearableExtender` gives you lots of flexibility in customizing wearable notifications.

Content Action

Certain notification actions are too important to be relegated to the second or third page. For instance, in a music player notification (see Figure 3-13), pausing and playing a song are fundamental features. You can set a content action to pause and play a song directly on the main notification page.

Once you add actions to your notification, you can set one of them as the content action, which will show up on the first page (see the bottom of Figure 3-20). Note that content actions no longer appear as buttons on their own screen. Also, the handheld notification is not affected by setting a content action. That is, content actions appear just like ordinary actions on handhelds.

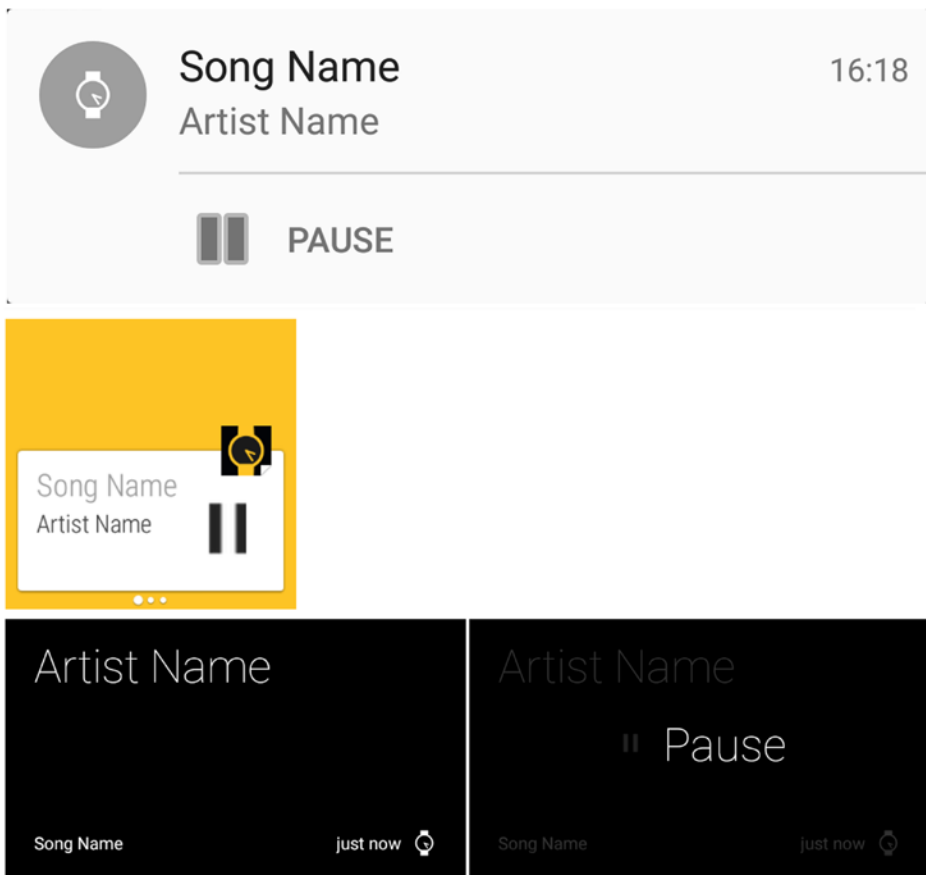


Figure 3-20. A notification with a content action. A handheld notification contains a single action (top). In Android Wear, a content action is triggered by tapping on the card of the main page (middle). In Glass, a content action appears as a menu item, just like a regular action (bottom)

Implementation

This section implements a notification with a content action.

Place the code from steps 1–3 where you want to issue the notification. In the sample project, we call this code in the `onContentActionClick` method.

Note When you start the example app, you will see several buttons that trigger different kinds of notifications. Tap on the last button (which is labeled *Content Action*) to trigger a notification with a content action.

1. Create two pause actions, one for the wearable and one for the handheld. These actions are identical except that they use different icons. Using different icons can be a good idea since wearable content actions appear on a white background.

```
PendingIntent pausePendingIntent = getActionFeedbackPendingIntent("You pressed pause", 0);
```

```
NotificationCompat.Action wearablePauseAction =
    new NotificationCompat.Action.Builder(R.drawable.ic_pause_dark,
        "Pause", pausePendingIntent)
        .build();
```

```
NotificationCompat.Action handheldPauseAction =
    new NotificationCompat.Action.Builder(android.R.drawable.ic_media_pause,
        "Pause", pausePendingIntent)
        .build();
```

2. Create a notification and set the content action of the `WearableExtender` to zero, which is the index of the pause action.

```
NotificationCompat.WearableExtender wearableExtender = new NotificationCompat.WearableExtender()
    .addAction(wearablePauseAction)
    .setContentAction(0);
```

```
Notification notification = new NotificationCompat.Builder(this)
    .setContentTitle("Song Name")
    .setContentText("Artist Name")
    .setSmallIcon(R.drawable.ic_stat_notify)
    .setContentIntent(getMainActivityPendingIntent())
    .setCategory(Notification.CATEGORY_TRANSPORT)
    .addAction(handheldPauseAction)
    .extend(wearableExtender)
    .build();
```

3. Issue the notification.

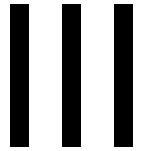
```
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, notification);
```

Content actions let you emphasize an important action in Android Wear. In contrast, defining a content action has no effect in Glass.

Summary

In this chapter, we learned to customize notifications for wearables by using `WearableExtender`. We enhanced wearable notifications with additional actions, pages, stacks, and voice actions without affecting the handheld notification. So far, all of the code we've written resides on the handheld app, and this approach is quick and effective for simple notifications on the wearable. For more sophisticated wearable apps, however, we need to create a separate app that runs on the wearable. This approach lets us create our own activities and services that run directly on wearables. In the next chapter, we'll learn to create wearable apps that run directly on Android Wear.

Part



Android Wear

Running Apps Directly on Android Wear

In chapters 2 and 3, we learned to build notifications on handheld apps and customize them for wearables. These notifications were issued in a handheld app and automatically shared with a paired wearable device. While notifications constitute an essential part of Android Wear, more involved applications require writing code that runs directly on a watch. Apps that run directly on Android Wear are powerful because they let you leverage most of the features of the Android SDK. For instance, you can build your own activities and services just as you would with a handheld app.

This chapter begins by demonstrating how to create an Android Wear app in Android Studio. Then we'll learn to create voice actions that let users start apps, and we'll cover a few fundamental UI widgets, including `CircledImageView` and `WearableListView`. The chapter ends with a complete example implementation of a timer.

The Android SDK in Android Wear

The majority of the Android SDK can be used in Android Wear apps with the exception of `print`, `appwidget`, `usb hardware`, `webkit`, and `network`. Note that Android Wear devices cannot directly access the internet and, as a result, classes such as `URLConnection` do not work as expected. However, Android Wear can access the internet indirectly through a paired handheld device.

There are also components that are not part of the Android SDK but are exclusive to Android Wear, namely the wearable UI library and the wearable data layer. The former contains a series of views/widgets designed for watches, and the latter allows wearable apps to communicate with handheld apps on a paired device. Chapter 5 elaborates on the wearable UI library and Chapter 6 covers the wearable data layer.

Before starting the next section, you should have an Android Wear device or emulator paired to your phone.

Creating a New Project

This section briefly explains how to use Android Studio to create a project with two modules, one for a handheld and the other for an Android Wear device. If you're already familiar with this process, consider skipping this section. Also note that the screen captures below were taken on a computer running Windows 7 with Android Studio 1.1 and your screens may look slightly different.

Note If you are a former Eclipse user, you may be confused by Android Studio's use of the terms 'project' and 'module.' A workspace in Eclipse is analogous to a project in Android Studio, and a project in Eclipse is analogous to a module in Android Studio. In other words, a workspace contains one or more projects in Eclipse, while a project contains one or more modules in Android Studio.

1. Click on File ► New Project to open the new project dialog and specify an Application name and a Package name for the project. I'll use the values shown in Figure 4-1.

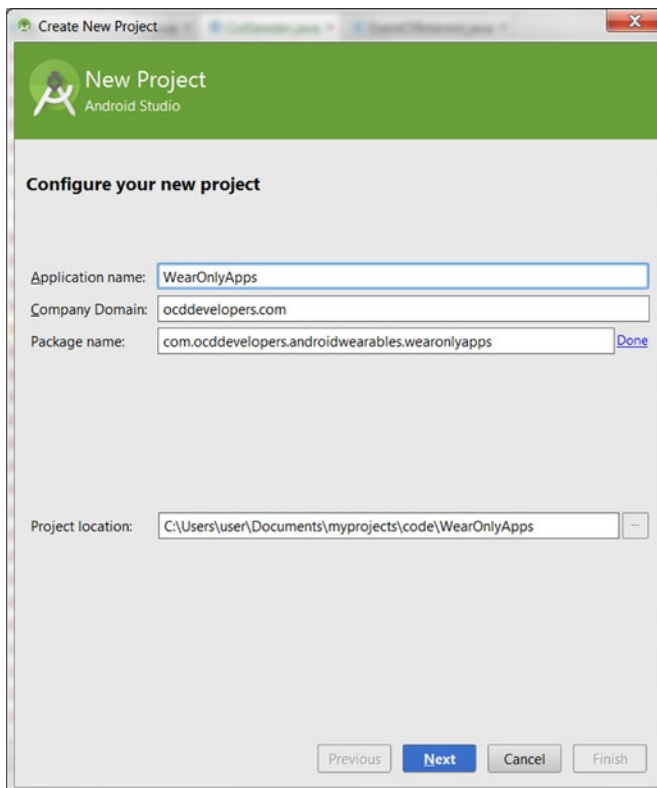


Figure 4-1. Configuring a project with a handheld and a wearable module

2. Specify that your app will run on two form factors: 'Phone and Tablet' and 'Wear,' as shown in Figure 4-2. Android Studio will create two modules: the 'mobile' module, which is configured for a handheld app and the 'wear' module, which is configured for a wearable app.

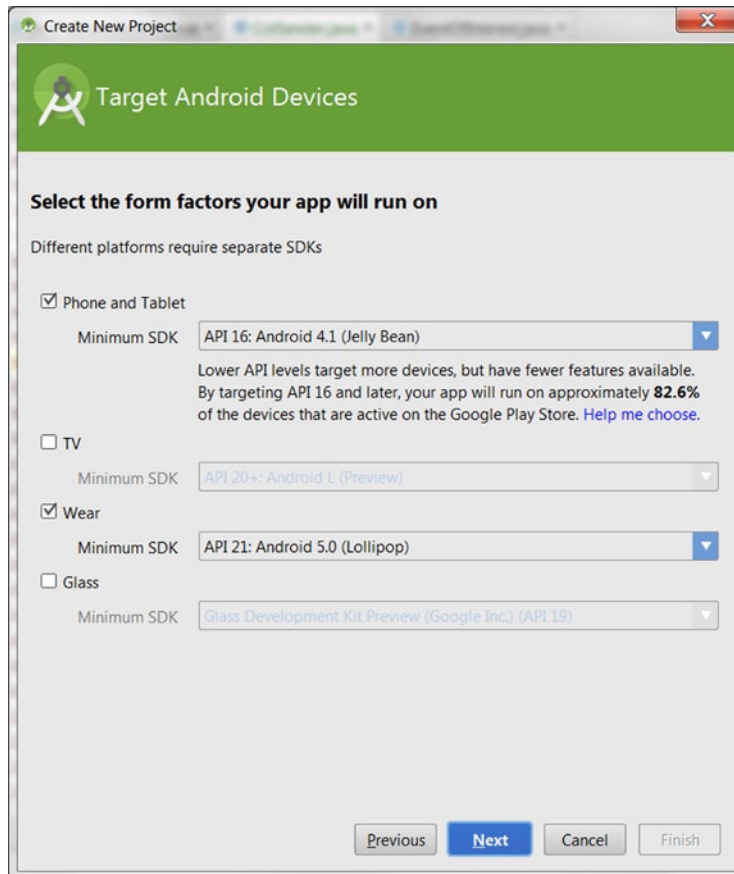


Figure 4-2. Selecting handheld (that is, phone and tablet) and wearable (that is, wear) form factors

3. For the purposes of this chapter, do not auto-generate any activities (see Figure 4-3). In subsequent examples, we'll create activities manually.

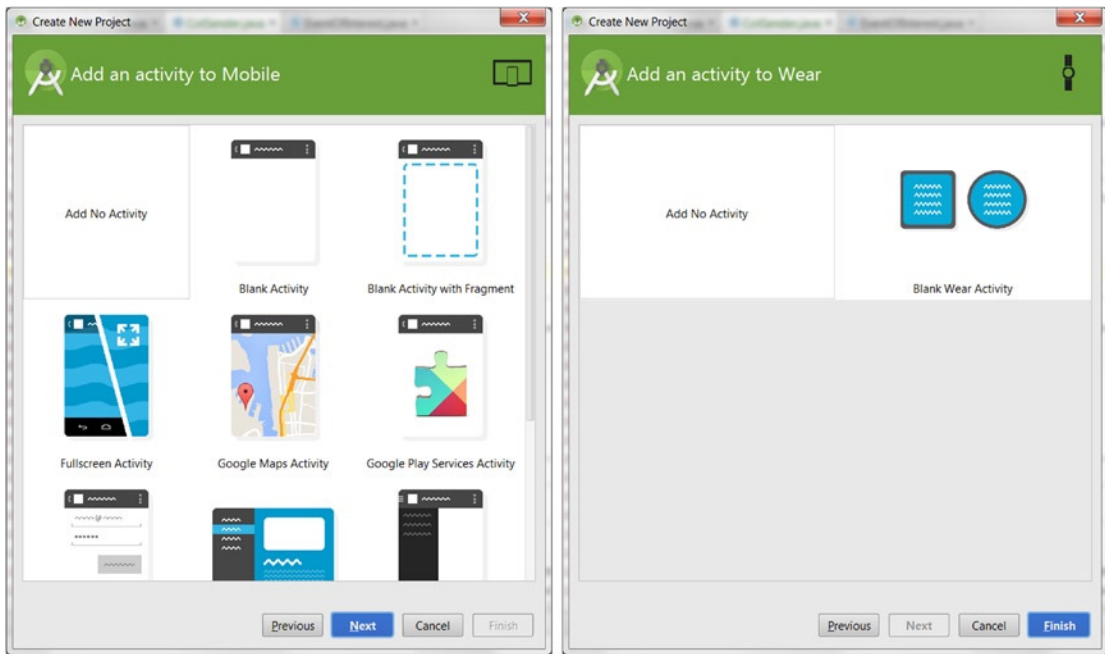


Figure 4-3. Do not auto-generate any activities for either mobile or wear modules

Although this section demonstrated how to simultaneously create both a handheld and a wearable app, the rest of this chapter will only use the wear app.

Starting Wear Apps

Starting apps in Android Wear is completely different than starting apps in handhelds, where users must simply tap on an app’s icon in the launcher screen. To start an app in Android Wear, the user must look at the device or tap on the screen to wake up the watch and either

- say “OK Google” to access the menu of available actions and speak the name of the desired action, or
- tap near the top of the watch’s screen to access the “OK Google” menu and then tap on the name of the action.

The list of actions is the same regardless of whether users access the menu with or without voice recognition, and the items in this menu are called voice actions even though they can be selected by tapping. There are two types of voice actions in Android Wear. *System-provided voice actions* are actions with predefined names that are built into the Android Wear platform, such as the “set a timer” command. These actions are available directly on the “OK Google” menu. In contrast, *app-provided voice actions* have custom names and can be declared by any app. Users can access the list of app-provided voice actions with the “Start...” voice action, which is typically located at the bottom of the “OK Google” menu.

When users trigger a voice action, it either starts an activity or a service. The way we declare a voice action depends on whether it’s system-provided or app-provided.

App-Provided Voice Actions

App-provided voice actions let users start an app from the “Start...” menu, which can be accessed with the “Start...” voice action located in the “OK Google” menu. Third party apps can declare app-provided voice actions with custom names. For example, the following declaration in `AndroidManifest` creates a voice action that starts `MainActivity` with the “Action Name” command:

```
<activity
  android:name=".MainActivity"
  android:label="Action Name" >
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

This declaration will likely seem familiar because it uses the same `intent-filter` as the top-level activity of a handheld app (that is, the one that appears in the launcher). In Android Wear, this intent filter denotes that the activity can be started with an app-provided voice action with the name specified by its label.

System-Provided Voice Actions

System-provided voice actions are intended for common user actions that can be handled by one or more apps, such as “take a note,” or “set an alarm.” When the user triggers a system-provided voice action, Android Wear starts an activity. If an app needs to start a service, it should call `startService(...)` and `finish()` in an activity’s `onCreate`.

Unlike app-provided voice actions, system-provided voice actions cannot have custom names. Instead, system-provided voice actions must have a name that belongs to a standard list that is managed by Google. The official documentation contains an up to date list of available system-provided voice actions (see <https://developer.android.com/training/wearables/apps/voice.html>). This list also contains the intent that is started with each system-provided voice action. To handle a system-provided voice action, declare an activity with an `intent-filter` that matches the desired system-provided voice action. For instance, use the following declaration to start `TimerActivity` when a user triggers the “set a timer” voice action:

```
<activity
  android:name=".TimerActivity"
  android:label="Timer Demo" >
  <intent-filter>
    <action android:name="android.intent.action.SET_TIMER" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

If multiple apps handle the same voice actions, users will be prompted to select the app that should handle the intent (see Figure 4-4). Once users make a selection, Android Wear will use the same app to handle future voice actions of the same type. However, users can still choose to handle a voice action with a different app from the Android Wear companion app.

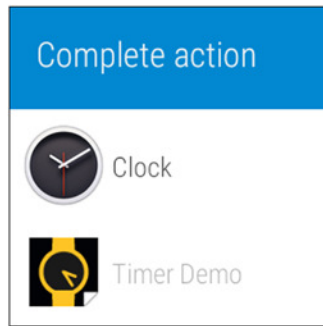


Figure 4-4. Selecting the activity to handle a system-provided voice action in Android Wear

The Example App

The example project for this chapter contains several launcher activities that demonstrate how to implement many aspects of a wearable app. For convenience, we place all of the examples within a chapter inside a single app, so you only have to import a single project per chapter. This setup has one undesired side effect. Say you start the launcher activity for example #1 and you cover the watch with your hand to turn the screen off. Then, you try to start the launcher activity for example #2. In this case, the activity for example #1 appears, and example #2 is nowhere to be seen. This behavior occurs because both activities belong to the same process.

When starting a launcher activity, Android checks to see if there are any existing activities on the task stack from the same process. If so, Android brings up that existing activity into the foreground. To avoid this behavior within the example app, we give each independent launcher activity a separate `taskAffinity`. Doing so creates each activity inside a task and prevents the situation outlined above. Keep in mind that most of your implementations will not need to set `taskAffinity`.

Example #1: Our First Wearable App

This example will demonstrate how to start an activity with an app-provided voice action. The activity displays a `TextView` in the center of the screen (see Figure 4-5).



Figure 4-5. The Hello Wear activity displays a TextView in the center of the screen

Note The source code for this example is located in the wear module of the WearOnlyApps project. Start the example with the “Start... Hello Wear” voice action.

1. Create a layout.

This layout shows the text “Hello Wear” in a TextView that is centered in the screen.

In res ► layout ► activity_hello.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:text="Hello Wear"
    android:layout_gravity="center"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

2. Declare the activity to be started with the “Hello Wear” app-provided voice action.

This is a standard launcher activity declaration with the exception of taskAffinity, which is only needed because this project uses multiple launcher activities that should be treated independently.

In AndroidManifest.xml:

```
<activity
    android:name=".HelloWearActivity"
    android:label="Hello Wear"
    android:taskAffinity="com.ocddevelopers.androidwearables.wearonlyapps.HelloTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3. Implement the activity.

In `HelloWearActivity.java`:

```
public class HelloWearActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello);
    }
}
```

Now that we've implemented the activity, you can run the app in Android Wear. In Android Studio, select the "wear" module from the dropdown menu shown in Figure 4-6 and then click on the *Run* button, which is the triangle to the right of the dropdown menu.

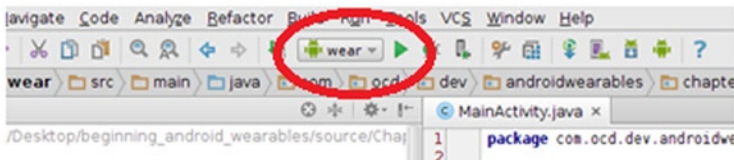


Figure 4-6. Running a wearable app from Android Studio

Android studio may launch the activity automatically since it's the app's entry point, but you should be able to start it again with the "Start... Hello Wear" voice action.

The Wearable UI Library

The rest of the projects in this chapter utilize the wearable UI library. To add this library as a dependency with Android Studio, open the project structure dialog from the File menu. Then, click on the "wear" module and on the "dependencies" tab. If not already present, add the `com.google.android.support:wearable` dependency. Depending on how you created the project, Android Studio may automatically add this library as a dependency.

Our First Wearable UI View: WearableListView

This wearable UI library contains several views designed specifically for Android Wear devices. Views designed for handholds are often difficult to use in wearables because they require precise gestures that users cannot comfortably perform on a tiny screen. A traditional `ListView`, for instance, should not be used with Android Wear because users would have a hard time selecting individual list items. Instead, we should use a view from the wearable UI library called `WearableListView`.

`WearableListView` lets users scroll up and down through a vertical list of items. Unlike a regular `ListView`, the nearest item automatically snaps to the center of the screen. Users select the item at the center of the screen by tapping near the middle of the screen.

WearableListView and RecyclerView

`WearableListView` extends `RecyclerView`, which is a more general implementation of `ListView` that is comparable to a traditional `ListView`'s `Adapter`. To avoid instantiating a new view every time a new list item becomes visible, `RecyclerView` does not discard views that move off the screen as the users scrolls. Instead, `RecyclerView` recycles these views and reuses them for the list items that scroll into the screen. Reusing views eliminates the overhead of constantly instantiating new ones.

The ViewHolder Pattern

When `RecyclerView` populates a child view with text, images, or other content, it calls `findViewById` to obtain a reference to the child view. Unfortunately, repeatedly calling `findViewById` takes time and can reduce the frame rate of a `RecyclerView`.

The view holder pattern, which was originally used while implementing standard `ListView` adapters, eliminates the overhead of repeatedly calling `findViewById` by storing references to the child views. These references are stored in a container called the `ViewHolder`. Using the view holder pattern in traditional `ListsViews` is a best practice that consists of creating a new class, which is typically called `ViewHolder`, to store the references to the child views (see <http://developer.android.com/training/improving-layouts/smooth-scrolling.html>). Although using a `ViewHolder` with traditional `ListsViews` is an optional best practice, `RecyclerView` and its subclasses such as `WearableListView` must be implemented with a `ViewHolder`.

The API of a `RecyclerView Adapter` utilizes `ViewHolders` directly and encourages you to use the view holder pattern.

Example #2: Implementing a List of Strings

This section demonstrates how to use `WearableListView` to implement the screen shown in Figure 4-7.

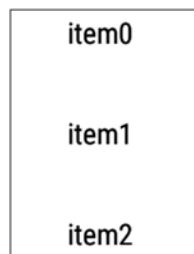


Figure 4-7. A sample `WearableListView` contains a single line of text per row

Note The source code for this example is located in the `wear` module of the `WearOnlyApps` project. Start the example with the “Start... Simple List Demo” voice action.

Extending WearableListView.Adapter

The `WearableListView.Adapter` class is responsible for instantiating the views used for every row in a `WearableListView` and for binding a row to the appropriate data. In this example, we'll extend `WearableListView.Adapter` in an inner class of `SimpleWearableListViewActivity` that is called `SampleAdapter`.

1. Declare and initialize member variables.

In `SimpleWearableListViewActivity.java`:

```
public static class SampleAdapter extends WearableListView.Adapter {
    private LayoutInflater mInflater;
    private String[] mSampleItems;

    public SampleAdapter(Context context, String[] sampleItems) {
        mInflater = LayoutInflater.from(context);
        mSampleItems = sampleItems;
    }
    ...
}
```

2. Create new views as needed.

Any time a `WearableListView` needs to create a new view for a row in the list, it calls the `onCreateViewHolder` method, which returns a `ViewHolder` that contains the inflated view. In this example, we'll inflate a built-in layout that only contains a `TextView`: `android.R.layout.simple_list_item_1`.

In `SimpleWearableListViewActivity.java`:

```
@Override
public WearableListView.ViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {
    View view = mInflater.inflate(android.R.layout.simple_list_item_1, viewGroup, false);
    return new WearableListView.ViewHolder(view);
}
```

3. Bind the appropriate data to each view as needed.

As users scroll, `WearableListView` determines the position (that is, index) of every visible row in the list. It then calls the `onBindViewHolder` method to load the appropriate data to a particular position.

The `ViewHolder` parameter has a public variable called `itemView` that contains the view used to instantiate the `ViewHolder`. In this example, we can simply cast the `itemView` to a `TextView`.

In `SimpleWearableListViewActivity.java`:

```
@Override
public void onBindViewHolder(WearableListView.ViewHolder viewHolder, int position) {
    TextView textView = (TextView) viewHolder.itemView;
    textView.setText(mSampleItems[position]);
}
```


4. Specify how many items are in the list.

In `SimpleWearableListViewActivity.java`:

```
@Override
public int getItemCount() {
    return mSampleItems.length;
}
```

Implementing SimpleWearableListViewActivity

This activity inflates a `WearableListView` and uses the `SampleAdapter` class from the previous step to display a list of strings.

1. Create a layout that contains `WearableListView`.

This layout is wrapped by `BoxInsetLayout`, which ensures that `WearableListView` appears correctly on both round and square watches. In round watches, `BoxInsetLayout` reduces the size of its child to a square that fits inside the round screen. In square watches, `BoxInsetLayout` gives its child view the entire size of the screen. Chapter 5 elaborates on `BoxInsetLayout` and other mechanisms to ensure compatibility with both square and round screens.

In `res > layout > activity_wlistview`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.wearable.view.WearableListView
        android:id="@+id/list"
        app:layout_box="left|bottom|right"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</android.support.wearable.view.BoxInsetLayout>
```

2. Declare the activity to be started with the “Simple List Demo” voice action.

This is a standard launcher activity declaration with the exception of `taskAffinity`, which is only needed because this project uses multiple launcher activities that should be treated independently.

In AndroidManifest.xml:

```
<activity
    android:name=".SimpleWearableListViewActivity"
    android:label="Simple List Demo"
    android:taskAffinity="com.ocdddevelopers.androidwearables.wearonlyapps.ListTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3. Declare and initialize member variables.

In this step, set the `WearableListView`'s adapter and click listener.

In `SimpleWearableListViewActivity.java`:

```
public class SimpleWearableListViewActivity extends Activity {
    private String[] mItems;
    private WearableListView mWearableListView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_wlistview);

        mItems = new String[] { "item0", "item1", "item2", "item3", "item4" };
        mWearableListView = (WearableListView) findViewById(R.id.list);
        mWearableListView.setAdapter(new SampleAdapter(this, mItems));
        mWearableListView.setOnClickListener(mClickListener);
    }
    ...
}
```

4. Create a `ClickListener`, which gets called when users tap on an item from the list.

In `SimpleWearableListViewActivity.java`:

```
private WearableListView.ClickListener mClickListener = new WearableListView.ClickListener() {
    @Override
    public void onClick(WearableListView.ViewHolder viewHolder) {
        int position = viewHolder.getPosition();
        Toast.makeText(SimpleWearableListViewActivity.this, "Tapped on item " + position,
            Toast.LENGTH_LONG).show();
    }

    @Override
    public void onTopEmptyRegionClick() {
        Toast.makeText(SimpleWearableListViewActivity.this, "Tapped on top empty region",
            Toast.LENGTH_LONG).show();
    }
};
```

The `onClick` method displays a `Toast` message when the user chooses an item from `WearableListView`. The `onTopEmptyRegionClick` is called when users tap on the empty region above the first item of a list.

Run the code on your device and make sure you understand how to implement this basic example before moving on to the next example.

Example #3: Implementing an Animated `WearableListView`

The previous example implemented a minimal `WearableListView` that only contained a string of text. However, most of the `WearableListView`s that you encounter within the system apps (such as setting a timer or alarm) contain a circular icon at the left of each row. The item in the center of the list has a large blue circle as its icon while other items have a smaller grey icon (see Figure 4-8). Moreover, when an item moves to the center of the list, its circular icon expands with an animation.



Figure 4-8. A `WearableListView` with a circular icon at the left of each row

This section implements this more elaborate `WearableListView`.

Note The source code for this example is located in the `wear` module of the `WearOnlyApps` project. Start the example with the “Start... Animated List Demo” voice action.

Create a Layout for Each Row

The appearance of every row in the `WearableListView` depends on whether it’s located at the center of a list (see Figure 4-8). The `WearableListItemLayout` class represents the layout of a single row in the list and draws the correct icon size and color. Additionally, it animates the change in icon size when an item enters or leaves the center position.

1. The radius of the icon is 16 dips if a row is in the center position and 10 dips otherwise.

In res ► values ► `dimens.xml`:

```
<dimen name="small_circle_radius">10dip</dimen>
<dimen name="big_circle_radius">16dip</dimen>
```

2. Create a layout for a single row.

The circular icon and text are contained within `WearableListItemLayout`, which is a subclass of `LinearLayout` that we'll implement shortly. We'll draw the circular icon with a `CircledImageView`, which is also part of the wearable UI library. `CircledImageView` creates a circle of a given color and radius (that is, `circle_color` and `circle_radius`, respectively). This view can display an image or icon within the circle, but this example does not utilize that functionality. We'll learn more about `CircledImageView` in chapter 5.

In res ► values ► `list_item.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<com.ocddevelopers.androidwearables.wearonlyapps.WearableListItemLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:orientation="horizontal"
  android:gravity="center_vertical"
  android:layout_width="match_parent"
  android:layout_height="80dip">

  <android.support.wearable.view.CircledImageView
    android:id="@+id/circle"
    app:circle_color="#c1c1c1"
    app:circle_radius="@dimen/small_circle_radius"
    app:circle_radius_pressed="@dimen/big_circle_radius"
    android:layout_gravity="center_vertical"
    android:layout_marginLeft="16dip"
    android:layout_marginRight="16dip"
    android:layout_width="20dip"
    android:layout_height="20dip" />

  <TextView
    android:id="@+id/name"
    android:gravity="center_vertical|left"
    android:layout_marginRight="16dip"
    android:fontFamily="sans-serif-condensed-light"
    android:lineSpacingExtra="-4sp"
    android:textColor="#434343"
    android:textSize="16sp"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"/>

</com.ocddevelopers.androidwearables.wearonlyapps.WearableListItemLayout>
```

3. With the layout from the previous step, we can implement `WearableListItemLayout`. In this step, declare its member variables.

The `WearableListView.OnCenterProximityListener` interface is used by `WearableListView` to notify the layout of whether it's located in the center of a list or not.

In `WearableListItemLayout.java`:

```
public class WearableListItemLayout extends LinearLayout
    implements WearableListView.OnCenterProximityListener {

    private CircledImageView mCircle;
    private TextView mName;
    private final float mFadedTextAlpha;
    private final int mUnselectedCircleColor, mSelectedCircleColor, mPressedCircleColor;
    private boolean mIsInCenter;
    private float mBigCircleRadius;
    private float mSmallCircleRadius;
    private ObjectAnimator mScalingDownAnimator;
    private ObjectAnimator mScalingUpAnimator;
    ...
}
```

4. Initialize member variables.

In `WearableListItemLayout.java`:

```
public WearableListItemLayout(Context context) {
    this(context, null);
}

public WearableListItemLayout(Context context, AttributeSet attrs) {
    this(context, attrs, 0);
}

public WearableListItemLayout(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);

    mFadedTextAlpha = 40 / 100f;
    mUnselectedCircleColor = Color.parseColor("#c1c1c1");
    mSelectedCircleColor = Color.parseColor("#3185ff");
    mPressedCircleColor = Color.parseColor("#2955c5");
    mSmallCircleRadius = getResources().getDimensionPixelSize(R.dimen.small_circle_radius);
    mBigCircleRadius = getResources().getDimensionPixelSize(R.dimen.big_circle_radius);

    // when expanded, the circle may extend beyond the bounds of the view
    setClipChildren(false);
}
```

When a circular icon is in a central position, it grows and extends outside of the dimensions of its view. The `setClipChildren(false)` call ensures that the circular icon can grow outside of the view's bounds without being clipped.

5. Obtain references to views and initialize animators.

We can find a reference to the layout's child views only after the layout has been inflated, which is why we do so in the `onFinishInflate` callback. Then, we initialize two `ObjectAnimators` to animate the radius of the circular icon as it grows or shrinks when a row enters or leaves the center of a list. An `ObjectAnimator` quickly modifies a property of an object according to the parameters of its animation. We're using the `ObjectAnimator.ofFloat` constructor, which takes three parameters:

- `Object target`: the object that contains the property to be animated. In this case, the `CircularImageView`.
- `String propertyName`: the name of the property of the target that should be animated. In this case, we want to animate `CircularImageView`'s circular radius, which is modified with the `setCircleRadius` method, so the property name is "circleRadius."
- `float... values`: one or more values that the animator will give the property over time. In this case, we only use one value, which is the value that the circular radius animates to.

In `WearableListItemLayout.java`:

```
@Override
protected void onFinishInflate() {
    super.onFinishInflate();
    mCircle = (CircledImageView) findViewById(R.id.circle);
    mName = (TextView) findViewById(R.id.name);

    mScalingUpAnimator = ObjectAnimator.ofFloat(mCircle, "circleRadius", mBigCircleRadius);
    mScalingUpAnimator.setDuration(150L);

    mScalingDownAnimator = ObjectAnimator.ofFloat(mCircle, "circleRadius",
    mSmallCircleRadius);
    mScalingDownAnimator.setDuration(150L);
}
```

6. When the row enters a central position, change the color of the circular icon and increase its size. Use an animation if the parameter indicates one is necessary.

In `WearableListItemLayout.java`:

```
@Override
public void onCenterPosition(boolean animate) {
    if(animate) {
        mScalingDownAnimator.cancel();
        if (!mScalingUpAnimator.isRunning() && mCircle.getCircleRadius() !=
        mBigCircleRadius) {
            mScalingUpAnimator.start();
        }
    } else {
        mCircle.setCircleRadius(mBigCircleRadius);
    }
}
```

```

    mName.setAlpha(1f);
    mCircle.setCircleColor(mSelectedCircleColor);
    mIsInCenter = true;
}

```

7. When the row exits a central position, change the color of the circular icon and decrease its size. Use an animation if the parameter indicates one is necessary.

In `WearableListItemLayout.java`:

```

@Override
public void onNonCenterPosition(boolean animate) {
    if(animate) {
        mScalingUpAnimator.cancel();
        if(!mScalingDownAnimator.isRunning() && mCircle.getCircleRadius()!=mSmallCircleRadius) {
            mScalingDownAnimator.start();
        }
    } else {
        mCircle.setCircleRadius(mSmallCircleRadius);
    }

    mName.setAlpha(mFadedTextAlpha);
    mCircle.setCircleColor(mUnselectedCircleColor);
    mIsInCenter = false;
}

```

8. When the user taps on the central list item, change the circular icon to a darker blue.

In `WearableListItemLayout.java`:

```

@Override
public void setPressed(boolean pressed) {
    super.setPressed(pressed);
    if(mIsInCenter && pressed) {
        mCircle.setCircleColor(mPressedCircleColor);
    }
    if(mIsInCenter && !pressed) {
        mCircle.setCircleColor(mSelectedCircleColor);
    }
}

```

Now that we have a layout for each row in the list, we can create an adapter to instantiate the layouts and populate them with data.

Implement WearableAdapter

The constructor of this class receives a list of strings that represent the labels of every row in the list. This class is very similar to the adapter of the previous example.

1. Create custom ViewHolder as an inner class of WearableAdapter.

This class stores a reference to a layout's TextView to avoid calling findViewById multiple times.

In WearableAdapter.java:

```
private static class ItemViewHolder extends WearableListView.ViewHolder {
    private TextView mItemTextView;

    public ItemViewHolder(View itemView) {
        super(itemView);
        mItemTextView = (TextView) itemView.findViewById(R.id.name);
    }
}
```

2. Declare and initialize member variables.

In WearableAdapter.java:

```
public class WearableAdapter extends WearableListView.Adapter {
    private List<String> mItems;
    private final LayoutInflater mInflater;

    public WearableAdapter(Context context, String[] items) {
        this(context, Arrays.asList(items));
    }

    public WearableAdapter(Context context, List<String> items) {
        mInflater = LayoutInflater.from(context);
        mItems = items;
    }
    ...
}
```

3. Inflate the list_item layout any time WearableListView asks for a new view. Store this view in an instance of the ItemViewHolder class, which we created in the previous step.

In WearableAdapter.java:

```
@Override
public WearableListView.ViewHolder onCreateViewHolder(ViewGroup viewGroup, int i) {
    return new ItemViewHolder(mInflater.inflate(R.layout.list_item, null));
}
```


4. `WearableListView` calls the `onBindViewHolder` to populate a view with the data for a given index of the list. In this method, obtain a `TextView` reference directly from the `ViewHolder` and set the appropriate text.

In `WearableAdapter.java`:

```
@Override
public void onBindViewHolder(WearableListView.ViewHolder viewHolder, int position) {
    ViewHolder itemViewHolder = (ViewHolder) viewHolder;
    TextView textView = itemViewHolder.mItemTextView;
    textView.setText(mItems.get(position));
    ((ViewHolder) viewHolder).itemView.setTag(position);
}
```

5. Specify the number of rows in the list.

In `WearableAdapter.java`:

```
@Override
public int getItemCount() {
    return mItems.size();
}
```

The final step is to create an activity that uses this adapter to populate a `WearableListView`.

Implementing `AnimatedListViewActivity`

This activity uses `WearableAdapter` to populate `WearableListView` with a list of strings.

1. Create a layout.

This class will use the `activity_wlistview.xml` layout from the previous example.

2. Declare the activity to be started with the “Animated List Demo” voice action.

This is a standard launcher activity declaration with the exception of `taskAffinity`, which is only needed because this project uses multiple launcher activities that should be treated independently.

In `AndroidManifest.xml`:

```
<activity
    android:name=".AnimatedListViewActivity"
    android:label="Animated List Demo"
    android:taskAffinity="com.ocddevelopers.androidwearables.wearonlyapps.AListTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3. Create `WearableListView` and set its adapter to an instance of `WearableAdapter`.

In `AnimatedListViewActivity.java`:

```
public class AnimatedListViewActivity extends Activity {
    private String[] mItems;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_wlistview);

        mItems = new String[] { "item0", "item1", "item2", "item3", "item4" };
        WearableListView wearableListView = (WearableListView) findViewById(R.id.list);
        wearableListView.setAdapter(new WearableAdapter(this, mItems));
        wearableListView.setOnClickListener(mClickListener);
    }
    ...
}
```

4. Display a message when a user taps on the row at the center of the list.

In `AnimatedListViewActivity.java`:

```
private WearableListView.ClickListener mClickListener = new WearableListView.ClickListener() {
    @Override
    public void onClick(WearableListView.ViewHolder viewHolder) {
        int position = viewHolder.getPosition();
        Toast.makeText(AnimatedListViewActivity.this, "Tapped on " + position,
            Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onTopEmptyRegionClick() {
    }
};
```

At this point, run the example on a watch and observe the behavior of the circular icon when you scroll and tap on the list.

The next section contains a more involved example that uses both notifications and a `WearableListView` to implement a timer.

Example #4: Creating a Timer App

The default timer app in Android Wear is pretty awesome. They say imitation is the sincerest form of flattery, so let's pay our compliments to the default timer app by replicating it in this example, which combines a system-provided voice action with a notification and a `WearableListView`. Note that this example is more involved than all previous examples.

Note The source code for this example is located in the wear module of the WearOnlyApps project. Start the example with the “Start... set a timer” voice action. If you are prompted to pick an activity to handle this intent, choose “Timer Demo.” A list will pop up to let you choose the duration of the timer. Alternatively, you may specify a duration directly in the voice action by saying, for instance, “set a timer for 25 minutes.” To revert the set timer voice action to use the default timer app, change the intent handler from the Android Wear companion app.

Overview of the Architecture

There are four main classes in the timer app.

- `TimerCompletedActivity` is started when the timer goes off. It displays a screen that indicates how much time has passed since the timer initially went off and causes the watch to vibrate every second (see Figure 4-9).

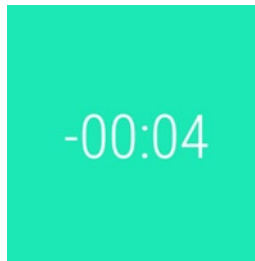


Figure 4-9. *TimerCompletedActivity vibrates the watch and shows the time elapsed since the timer went off*

- `TimerUtil` contains methods to create, pause, and stop a timer. Creating a timer involves building a notification that shows a countdown timer and scheduling an alarm that will start `TimerCompletedActivity` when the timer goes off. Figure 4-10 shows the notification.

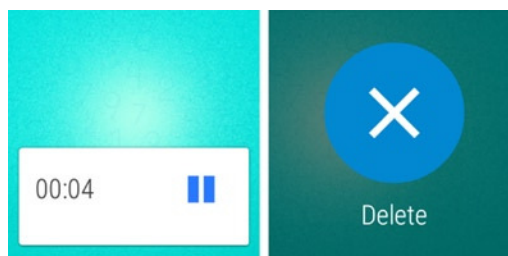


Figure 4-10. *The timer notification displays the pause/resume action as its content action. Additionally, it contains an action labeled “Delete” that stops the timer*

- `TimerActivity` is started in response to the “set a timer” voice action. If the user provides a duration in the voice action (for instance, “set a timer for 5 minutes”) it schedules the timer and finishes. If the user does not provide a duration, it prompts the user for a duration and then schedules the timer and finishes (see Figure 4-11).

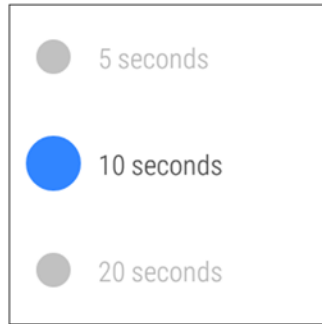


Figure 4-11. If the “set a timer” voice action does not specify a duration, `TimerActivity` displays a list that allows users to input this duration

- `CommandReceiver` is a `BroadcastReceiver` that pauses, resumes, or stops the timer when a user taps on a respective action in the timer notification. Tapping on one of these actions triggers a `PendingIntent` that invokes `CommandReceiver` with the appropriate action specified as the intent’s action. This class starts the `TimerCompletedActivity` when Android’s `AlarmManager` triggers a `PendingIntent` when the timer goes off. It delegates other actions to `TimerUtil`.

Before implementing `TimerUtil`, let’s see how to include a countdown timer within a notification.

Using a Chronometer in a Notification

Although we could create a countdown timer in a notification by updating the content text every second with a new countdown, it would be inefficient and tedious to implement. Fortunately, this method is unnecessary thanks to the `setUsesChronometer` method, which is found in `NotificationCompat.Builder` (see <http://developer.android.com/reference/android/support/v4/app/NotificationCompat.Builder.html#setUsesChronometer%28boolean%29> for the official documentation).

Notifications built with `setUsesChronometer(true)` will display a stopwatch that counts up from the time given to the `setWhen` method, which uses the same units as `System.currentTimeMillis`. Unlike a chronometer, a timer does not count up, but counts down. An undocumented trick is that giving `setWhen` a time that’s in the future creates a timer in Android Wear devices. This trick does not yet work with Android handhelds, so make sure you only use it in wearables.

We’ll use this technique below.

Implementing TimerUtil

TimerUtil is responsible for creating and removing the timer notification, which displays the time remaining and provides actions to let users pause, resume, and stop the timer.

1. Declare member variables.

In TimerUtil.java:

```
public class TimerUtil {
    private static final int NOTIFICATION_ID = 1;
    public static final String PREFS_PREV_START_TIME = "prev_start_time";
    public static final String PREFS_TIMER_DURATION = "timer_duration";
    public static final String PREFS_IS_PAUSED = "is_paused";
    public static final String PREFS_TIME_ELAPSED = "time_elapsed";
    ...
}
```

2. Create a new timer.

This method creates the notification for a timer and schedules an alarm to make the timer go off after the specified duration. Additionally, this method stores several values that are relevant to the timer, such as its duration. The `setTimerAlarm` and `createTimerNotification` methods are implemented in the following steps.

In TimerUtil.java:

```
public static void createNewTimer(Context context, long duration) {
    long startTime = System.currentTimeMillis();
    SharedPreferences.Editor editor = PreferenceManager.getDefaultSharedPreferences
        (context).edit();
    editor.putLong(PREFS_PREV_START_TIME, SystemClock.elapsedRealtime());
    editor.putLong(PREFS_TIMER_DURATION, duration);
    editor.putBoolean(PREFS_IS_PAUSED, false);
    editor.putLong(PREFS_TIME_ELAPSED, 0L);
    editor.commit();

    setTimerAlarm(context, startTime+duration);
    createTimerNotification(context, startTime+duration, false);
}
```

3. Set an alarm to trigger the timer.

This method uses `AlarmManager` to schedule an alarm to trigger an activity when the timer goes off. Read the documentation on `AlarmManager` for more details on scheduling alarms (<http://developer.android.com/reference/android/app/AlarmManager.html>). In this case, `AlarmManager` triggers a `PendingIntent` that starts `TimerCompletedActivity` when the timer goes off. The `alarmTime` parameter is the time at which the timer is finished and uses the same time scale as `System.currentTimeMillis`.

In `TimerUtil.java`:

```
private static void setTimerAlarm(Context context, long alarmTime) {
    Intent completedIntent = CommandReceiver.makeTriggerAlarmIntent(alarmTime);
    PendingIntent completedPendingIntent = PendingIntent.getBroadcast(context, 0,
        completedIntent, PendingIntent.FLAG_CANCEL_CURRENT);

    AlarmManager alarmManager = (AlarmManager) context.getSystemService(Context.ALARM_SERVICE);
    alarmManager.setExact(AlarmManager.RTC_WAKEUP, alarmTime, completedPendingIntent);
}
```

4. Implement `createTimerNotification`.

This method creates a notification that displays the time remaining and lets the user pause/resume, and stop the timer. If the timer is currently paused, the notification shows the resume action as its content action. Otherwise, it shows the pause action as its content action. Note that the notification uses `WearableExtender` to add a background, and that it hides the app icon.

The `PendingIntents` for the actions trigger a broadcast that is handled by `CommandReceiver`, which we'll implement in the next section.

In `TimerUtil.java`:

```
private static void createTimerNotification(Context context, long when, boolean isPaused) {
    Intent pauseIntent = CommandReceiver.makePauseTimerIntent();
    PendingIntent pausePendingIntent = PendingIntent.getBroadcast(context, 0,
        pauseIntent, PendingIntent.FLAG_CANCEL_CURRENT);

    Intent stopIntent = CommandReceiver.makeStopTimerIntent();
    PendingIntent stopPendingIntent = PendingIntent.getBroadcast(context, 0,
        stopIntent, PendingIntent.FLAG_CANCEL_CURRENT);

    Bitmap background;
    int pauseResumeIcon;
    String pauseResumeTitle;

    // if paused, show resume action. otherwise, show pause action
    if(isPaused) {
        pauseResumeIcon = R.drawable.ic_stopwatch_play;
        pauseResumeTitle = "Resume";
        background = BitmapFactory.decodeResource(context.getResources(),
            R.drawable.bg_timer_stopped);
    } else {
        pauseResumeIcon = R.drawable.ic_stopwatch_pause;
        pauseResumeTitle = "Pause";
        background = BitmapFactory.decodeResource(context.getResources(),
            R.drawable.bg_timer_running);
    }
}
```

```

// create notification
NotificationCompat.WearableExtender wearableExtender =
    new NotificationCompat.WearableExtender()
        .setBackground(background)
        .setContentAction(0)
        .setHintHideIcon(true);

NotificationCompat.Builder timerNotificationBuilder =
    new NotificationCompat.Builder(context)
        .setSmallIcon(R.drawable.ic_launcher)
        .setOngoing(true)
        .addAction(pauseResumeIcon, pauseResumeTitle, pausePendingIntent)
        .addAction(R.drawable.ic_full_cancel, "Delete", stopPendingIntent)
        .setPriority(Notification.PRIORITY_HIGH)
        .setCategory(Notification.CATEGORY_ALARM)
        .extend(wearableExtender);

// if paused, set content title and content text. otherwise, show timer countdown
if(isPaused) {
    int timerRemaining = (int)(when - System.currentTimeMillis())/1000;
    String time = String.format(Locale.US, "%02d:%02d", timerRemaining / 60,
        timerRemaining % 60);
    timerNotificationBuilder.setContentTitle(time).setContentText("");
} else {
    timerNotificationBuilder.setUsesChronometer(true).setWhen(when);
}

// issue notification
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(context);
notificationManager.notify(NOTIFICATION_ID, timerNotificationBuilder.build());
}

```

5. Implement stopTimer.

This method cancels the timer by stopping the alarm and removing the notification.

In TimerUtil.java:

```

public static void stopTimer(Context context) {
    removeTimerAlarm(context);
    removeTimerNotification(context);
}

```

6. Implement removeTimerAlarm.

This method cancels the alarm that starts TimerCompletedActivity.

In TimerUtil.java:

```

private static void removeTimerAlarm(Context context) {
    Intent alarmCompletedIntent = CommandReceiver.makeTriggerAlarmIntent(0L);
    PendingIntent alarmCompletedPendingIntent = PendingIntent.getBroadcast(context, 0,
        alarmCompletedIntent, PendingIntent.FLAG_NO_CREATE);
}

```

```
if(alarmCompletedPendingIntent != null) {
    AlarmManager alarmManager = (AlarmManager)context.getSystemService
        (Context.ALARM_SERVICE);
    alarmManager.cancel(alarmCompletedPendingIntent);
    alarmCompletedPendingIntent.cancel();
}
}
```

7. Implement `removeTimerNotification`.

This method removes the notification from the context stream.

In `TimerUtil.java`:

```
private static void removeTimerNotification(Context context) {
    NotificationManagerCompat.from(context).cancel(NOTIFICATION_ID);
}
```

8. Implement `pauseTimer`.

This method pauses the timer, issues a notification that indicates the timer is paused, and saves all the state needed to resume the timer. When `pauseTimer` is called again, it resumes the timer.

In `TimerUtil.java`:

```
public static void pauseTimer(Context context) {
    SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(context);
    long now = SystemClock.elapsedRealtime();
    long prevStartTime = prefs.getLong(PREFS_PREV_START_TIME, 0L);
    long elapsed = prefs.getLong(PREFS_TIME_ELAPSED, 0L);
    long duration = prefs.getLong(PREFS_TIMER_DURATION, 0L);
    boolean isPaused = prefs.getBoolean(PREFS_IS_PAUSED, false);

    if(isPaused) {
        // resume timer
        long timerDueTimeMillis = System.currentTimeMillis() + duration - elapsed;
        setTimerAlarm(context, timerDueTimeMillis);
        createTimerNotification(context, timerDueTimeMillis, false);

        SharedPreferences.Editor editor = prefs.edit();
        editor.putLong(PREFS_PREV_START_TIME, now);
        editor.putBoolean(PREFS_IS_PAUSED, false);
        editor.commit();
    } else {
        // pause timer
        elapsed = elapsed + now - prevStartTime;
        removeTimerAlarm(context);
        createTimerNotification(context, duration-elapsed + System.currentTimeMillis(), true);
    }
}
```



```

        SharedPreferences.Editor editor = prefs.edit();
        editor.putLong(PREFS_TIME_ELAPSED, elapsed);
        editor.putBoolean(PREFS_IS_PAUSED, true);
        editor.commit();
    }
}

```

TimerUtil contains all the functionality needed to create a timer. The timer notification contains actions that start broadcasts handled by `CommandReceiver`, which we'll implement next.

Implementing CommandReceiver

When the timer goes off, an alarm triggers a `PendingIntent` that starts a broadcast that `CommandReceiver` handles. Similarly, when users tap on an action to pause/resume or stop the timer, the system triggers a `PendingIntent` that starts a broadcast that `CommandReceiver` handles.

1. Add the wake lock permission.

In `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
```

2. In the manifest, declare `CommandReceiver` with an intent-filter that accepts actions for triggering the alarm, pausing/resuming the timer, and stopping the timer.

Note that an intent-filter with multiple actions matches intents that contain any of those actions.

In `AndroidManifest.xml`:

```

<receiver android:name=".CommandReceiver" >
    <intent-filter>
        <action android:name=
            "com.ocddevelopers.androidwearables.wearonlyapps.action.TRIGGER_ALARM" />
        <action android:name=
            "com.ocddevelopers.androidwearables.wearonlyapps.action.PAUSE_TIMER" />
        <action android:name="com.ocddevelopers.androidwearables.wearonlyapps.action.STOP_TIMER" />
    </intent-filter>
</receiver>

```

3. Declare constants and variables.

In `CommandReceiver.java`:

```
public class CommandReceiver extends BroadcastReceiver {
    private static final String ACTION_TRIGGER_ALARM =
        "com.ocddevelopers.androidwearables.wearonlyapps.action.TRIGGER_ALARM";
    private static final String ACTION_PAUSE_TIMER =
        "com.ocddevelopers.androidwearables.wearonlyapps.action.PAUSE_TIMER";
    private static final String ACTION_STOP_TIMER =
        "com.ocddevelopers.androidwearables.wearonlyapps.action.STOP_TIMER";
    private static final String EXTRA_START_TIME = "start_time";
    private static PowerManager.WakeLock mWakeLock;
    ...
}
```

4. Provide public methods that generate the actions needed to trigger, pause/resume, or stop the timer.

In `CommandReceiver.java`:

```
public static Intent makeTriggerAlarmIntent(long startTime) {
    Intent intent = new Intent(ACTION_TRIGGER_ALARM);
    intent.putExtra(EXTRA_START_TIME, startTime);
    return intent;
}

public static Intent makePauseTimerIntent() {
    return new Intent(ACTION_PAUSE_TIMER);
}

public static Intent makeStopTimerIntent() {
    return new Intent(ACTION_STOP_TIMER);
}
```

5. When a broadcast is started, check the intent's action to determine what command should be executed.

Delegate the pause/resume and stop actions to `TimerUtil`, and start `TimerCompletedActivity` when the timer goes off. The `startWakefulActivity` method ensures that the activity gets started even when the watch is asleep, as explained in the next step.

In `CommandReceiver.java`:

```
@Override
public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();
    if(ACTION_TRIGGER_ALARM.equals(action)) {
        Intent completedIntent = new Intent(context, TimerCompletedActivity.class);
        completedIntent.putExtra(TimerCompletedActivity.EXTRA_START_TIME,
            intent.getLongExtra(EXTRA_START_TIME, 0L));
        completedIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        CommandReceiver.startWakefulActivity(context, completedIntent);
    }
}
```

```

    } else if(ACTION_PAUSE_TIMER.equals(action)) {
        TimerUtil.pauseTimer(context);
    } else if(ACTION_STOP_TIMER.equals(action)) {
        TimerUtil.stopTimer(context);
    }
}

```

6. Implement startWakefulActivity.

In `CommandReceiver.java`:

`AlarmManager` guarantees that a `BroadcastReceiver`'s `onReceive` will be called in its entirety before the system goes back to sleep.

The problem is that `AlarmManager` likes to save battery and only guarantees that the device will be awake during the `BroadcastReceiver`'s `onReceive`. In other words, the device could go to sleep after `startActivity` is called but before the Activity is actually started. The user wouldn't realize that the timer went off until he wakes the device up. This is unacceptable.

Consequently, when starting an activity inside `onReceive`, the system may go back to sleep before the activity actually starts. The `startWakefulActivity` method solves this issue by acquiring a `WakeLock` that ensures the system stays awake until the activity is started.

```

public static void startWakefulActivity(Context context, Intent intent) {
    PowerManager pm = (PowerManager)context.getSystemService(Context.POWER_SERVICE);
    mWakeLock = pm.newWakeLock(PowerManager.SCREEN_BRIGHT_WAKE_LOCK |
        PowerManager.ACQUIRE_CAUSES_WAKEUP |
        PowerManager.ON_AFTER_RELEASE, "wakeful activity lock");
    mWakeLock.setReferenceCounted(false);
    mWakeLock.acquire(60 * 1000);

    context.startActivity(intent);
}

```

7. Implement completeWakefulIntent.

Once an activity is started wakefully, it no longer needs to hold the `WakeLock`. An activity should therefore call the `CommandReceiver.completeWakefulIntent` method in its `onResume` callback.

In `CommandReceiver.java`:

```

public static void completeWakefulIntent() {
    if(mWakeLock != null) {
        mWakeLock.release();
        mWakeLock = null;
    }
}

```

As an experiment, try starting the `TimerCompletedActivity` with `startActivity` instead of `CommandReceiver.startWakefulActivity`. Then, start a timer with a duration of 20 seconds and put your watch to sleep before the timer goes off. There is a big chance that the timer will not go off after 20 seconds. Instead, the timer may go off as soon as you wake up the watch.

The next section implements `TimerCompletedActivity`.

Implementing `TimerCompletedActivity`

This activity should be started when the timer goes off and repeatedly vibrates the watch while displaying the time elapsed since the timer went off.

1. Add the vibrate permission.

In `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.VIBRATE" />
```

2. Declare the activity.

In `AndroidManifest.xml`:

```
<activity android:name=".TimerCompletedActivity" />
```

3. Create a layout.

This layout contains a `TextView` that displays the time since the timer went off.

In `res > layout > activity_timer_completed.xml`:

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:background="#ff1ce8b5"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/time_since_completed"
        android:textColor="#ffffffff"
        android:textSize="44sp"
        android:fontFamily="sans-serif-condensed-light"
        android:layout_gravity="center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</FrameLayout>
```

4. Declare member variables.

In `TimerCompletedActivity.java`:

```
public class TimerCompletedActivity extends Activity {
    public static final String EXTRA_START_TIME = "start_time";
    private static final long VIBRATION_DURATION = 200L;
    private static final int DELAY_MILLIS = 1000;
    private static final int MILLIS_PER_SECOND = 1000;
    private static final int MAX_DURATION_SECONDS = (int)TimeUnit.MINUTES.toSeconds(1);
    private TextView mTimeSinceCompleted;
    private Handler mHandler;
    private Vibrator mVibrator;
    private long mTimerCompletionTime;
    ...
}
```

5. Initialize variables and start vibrating the phone in the `onCreate` method.

The `keep screen on` flag ensures that the watch does not go to sleep until this activity is dismissed. If the required start time extra is not found, throw a `RuntimeException`.

In `TimerCompletedActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_timer_completed);

    // cancel timer notification
    NotificationManagerCompat.from(this).cancel(1);

    mTimeSinceCompleted = (TextView)findViewById(R.id.time_since_completed);
    mHandler = new Handler();
    mVibrator = (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);

    // start playing alarm
    mTimerCompletionTime = getIntent().getLongExtra(EXTRA_START_TIME, -1);
    if(mTimerCompletionTime >= 0) {
        mRunnable.run();
    } else {
        throw new RuntimeException(
            "TimerCompletedActivity requires the timer's start time extra");
    }
}
```

6. Once the activity is started, release the `WakeLock` that was acquired by `CommandReceiver`.

As we saw in the previous section, `CommandReceiver` acquires a `WakeLock` before starting this activity. The `WakeLock` ensures that the watch does not go to sleep before the activity is able to start.

In `TimerCompletedActivity.java`:

```
@Override
protected void onResume() {
    super.onResume();
    CommandReceiver.completeWakefulIntent();
}
```

7. Vibrate the watch every second and update the time since completion text.

In `TimerCompletedActivity.java`:

```
private Runnable mRunnable = new Runnable() {
    @Override
    public void run() {
        int deltaSeconds = (int)(System.currentTimeMillis() -
            mTimerCompletionTime)/MILLIS_PER_SECOND;
        String delta = DateUtils.formatElapsedTime(deltaSeconds);
        mTimeSinceCompleted.setText("-" + delta);
        mVibrator.vibrate(VIBRATION_DURATION);

        // buzz alarm once a second for no more than 1 minute
        if(deltaSeconds < MAX_DURATION_SECONDS) {
            mHandler.postDelayed(mRunnable, DELAY_MILLIS);
        }
    }
};
```

8. Stop vibrating the watch when a user dismisses the activity.

In `TimerCompletedActivity.java`:

```
@Override
protected void onPause() {
    mHandler.removeCallbacks(mRunnable);
    super.onPause();
}
```

Implementing TimerActivity

When the “Set a timer” system-provided voice action starts `TimerActivity`, it may contain an extra named `AlarmClock.EXTRA_LENGTH` that contains the duration of the timer if it was given as part of the voice action (for instance, “Set a timer for 20 minutes”). If the timer was not started with a duration, `TimerActivity` displays a list that asks the user for a duration. In either case, `TimerActivity` starts the timer with the specified duration and then automatically closes.

1. Create a layout.

This class will use the `activity_wlistview.xml` layout from the previous example.

2. Declare the activity to be started with the system-provided voice action for a timer.

This declaration indicates that `TimerActivity` should be started with the “Set timer” voice action. Note that the `taskAffinity` parameter is only needed because this project uses multiple launcher activities that should be treated independently.

```
<activity
    android:name=".TimerActivity"
    android:label="Timer Demo"
    android:taskAffinity="com.ocddevelopers.androidwearables.wearonlyapps.TimerTask">
    <intent-filter>
        <action android:name="android.intent.action.SET_TIMER" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

3. Initialize variables.

In `TimerActivity.java`:

```
public class TimerActivity extends Activity {
    private static final int[] TIMER_PRESETS_DURATION_SECS = { 5, 10, 20, 30, 40, 50, 60, 120,
        180, 240, 300, 360, 420, 480, 540, 600, 900, 1800, 2700, 3600, 5400 };
    private static final String[] TIMER_PRESETS_LABEL = new String[] {
        "5 seconds", "10 seconds", "20 seconds", "30 seconds", "40 seconds", "50 seconds",
        "1 minute", "2 minutes", "3 minutes", "4 minutes", "5 minutes",
        "6 minutes", "7 minutes", "8 minutes", "9 minutes", "10 minutes",
        "15 minutes", "30 minutes", "45 minutes", "1 hour", "1.5 hours"
    };
    ...
}
```

4. If the duration of the timer is available as an extra, create the timer and finish. Otherwise, populate `WearableListView` with the labels from `TIMER_PRESETS_LABEL`.

Note that this step uses the `WearableAdapter` class from a previous example.

In `TimerActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_wlistview);

    int timerDuration = getIntent().getIntExtra(AlarmClock.EXTRA_LENGTH, -1);
    if(timerDuration > 0) {
        TimerUtil.createNewTimer(this, timerDuration*1000L);
        finish();
    }
}
```

```

    } else {
        // of EXTRA_LENGTH is not available, then we must prompt the user for a timer duration
        WearableListView wearableListView = (WearableListView)findViewById(R.id.list);
        WearableAdapter timerDurationAdapter = new WearableAdapter(this,
            TIMER_PRESETS_LABEL);
        wearableListView.setAdapter(timerDurationAdapter);
        wearableListView.setOnClickListener(mClickListener);
    }
}

```

5. When a user selects a duration for the timer from the list, create the timer and finish the activity.

In `TimerActivity.java`:

```

private WearableListView.ClickListener mClickListener = new WearableListView.ClickListener() {
    @Override
    public void onClick(WearableListView.ViewHolder viewHolder) {
        // once the user picks a duration, create timer alarm/notification and exit
        int position = viewHolder.getPosition();
        int duration = TIMER_PRESETS_DURATION_SECS[position];
        TimerUtil.createNewTimer(TimerActivity.this, duration*1000L);
        finish();
    }

    @Override
    public void onTopEmptyRegionClick() {
    }
};

```

Test the code both with and without specifying the time as part of the voice command (for example, “set a timer for 10 minutes” and “set a timer,” respectively). As an exercise, try adding an action to restart the timer, just like the one in the default timer app.

Summary

We went over how to create and run Android Wear projects in Android Studio. Then we learned how to start apps with app-provided and system-provided voice actions. Afterwards, we covered `WearableListViews` and ended the chapter by implementing a timer app that combined many of the skills we have learned so far. In the next chapter, we’ll learn to use additional widgets from the wearable UI library and we’ll learn to create custom notification layouts.

Android Wear User Interface Essentials

Although we can use the Android SDK to build interfaces for Android Wear, its widgets and views were designed for handheld devices and are often not effective on wearables. A `ListView`, for instance, should not be used in Android Wear because tapping on a list item requires precise gestures that are difficult to make on a small screen. The design principles for Android Wear (see <https://developer.android.com/design/wear/principles.html>) remind us to keep user interactions to a minimum and to rely exclusively on coarse gestures. Thankfully, the wearable UI library contains several widgets and views designed specifically for Android Wear, such as `WearableListView`, which allows users to select an item from a list using coarse gestures (see Chapter 4).

In this chapter, we'll learn to use several widgets from the wearable UI library by implementing three example apps. The first app lets users review vocabulary words directly on their watches, the second app demonstrates how to implement confirmation animations and several views, and the third app tracks information that would be useful to a runner, such as speed, distance, and calories. For the purposes of this chapter, this latter app will only display simulated information and will not collect GPS data. The example apps will demonstrate how to use the following widgets and views:

first app: `CardFragment`, `GridViewPager`

second app: `CircledImageView`, `DelayedConfirmationView`, `WatchViewStub`, `ConfirmationActivity`, `GridViewPager`

third app: `BoxInsetLayout`, `GridLayout`, custom notifications

The next two sections discuss `CardFragment` and `GridViewPager`, which we'll use to implement the first example app.

Using CardFragment

Android Wear's context stream displays the content of each notification inside a card layout, as shown in Figure 5-1.



Figure 5-1. The content of a notification is displayed in a card layout

A card layout is a white rectangle that casts a shadow on the background and separates the content of a page from its background. The `CardFragment` class, which is part of the wearable UI library, displays its content on a card layout. Although we could implement a card layout without using the wearable UI library, its appearance and behavior would not necessarily be consistent with the cards used in the rest of the platform. Using card layouts with a consistent appearance makes it easy for users to figure out how to use a card: since it looks like any other card, it must therefore also behave like any other card.

In addition to applying a card-shaped background to your content, `CardFragment` makes the cards expandable and vertically scrollable. The height of the card will automatically adjust itself to the height of its content. If the content is shorter than the height of the screen, the card will only occupy part of the screen. If the content is taller than the height of the screen, the card will expand itself and let the user scroll up and down to see the entire content.

Although creating an instance of `CardFragment` is straightforward, it has several options that can be a little difficult to understand.

The Card Gravity Property

When the card is smaller than the height of the screen, the card gravity determines whether it's aligned with the top or the bottom of the screen (see Figure 5-2).

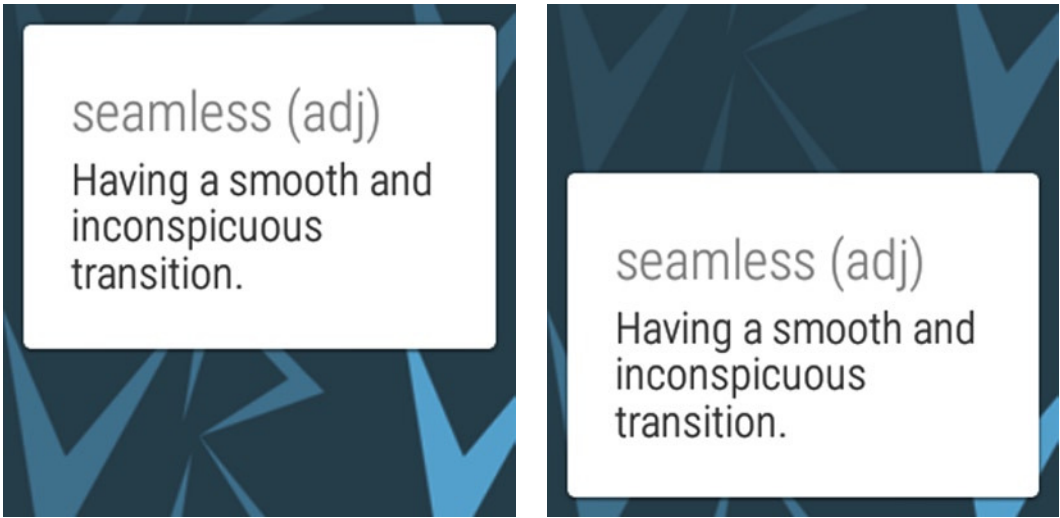


Figure 5-2. A CardFragment's gravity determines whether the layout is aligned with the top or bottom of the screen. A card with a gravity of Gravity.TOP (left), and a card with a gravity of Gravity.BOTTOM (right)

To set the card gravity, call

```
cardFragment.setCardGravity(int gravity);
```

(where the gravity parameter is either Gravity.TOP or Gravity.BOTTOM.)

The Card Expansion Property

This setting determines how a CardFragment handles cards that are too tall to fit in the screen. If expansion is disabled, the content that falls outside of the screen's bounds is cut off and cannot be accessed. If expansion is enabled, users can see the content that falls outside of the screen's bounds by scrolling up or down.

To enable expansion, call

```
cardFragment.setExpansionEnabled(true);
```

The Expansion Direction Property

If the CardFragment's expansion is enabled, the expansion direction determines whether the card expands up or down. If the card expands up, then the CardFragment initially shows the bottom of the content and the user can scroll up to see the rest. That is, the user will initially see the screen to the right of Figure 5-3. If the card expands down, the CardFragment initially shows the top of the content, and the user can scroll down to see the rest. That is, the user would initially see the screen in the middle of Figure 5-3.

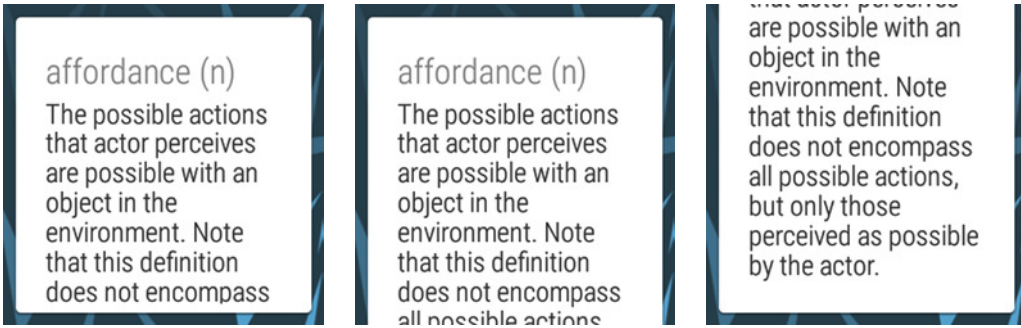


Figure 5-3. A `CardFragment`'s expansion determines how it handles content that falls outside of the screen. When expansion is disabled, any content that does not fit in the card is clipped and cannot be accessed (left). When expansion is enabled, the entire content of the card can be accessed by scrolling up or down (middle and right)

To set the expansion direction, call

```
cardFragment.setExpansionDirection(direction);
```

(where `direction` is either `CardFragment.EXPAND_UP` or `CardFragment.EXPAND_DOWN`.)

The Expansion Factor Property

When a `CardFragment`'s expansion is enabled, there is a limit as to how much the card can expand. The maximum height of the card is specified by the `expansion_factor`. An `expansion_factor` of two, for instance, means that the maximum height of the card is two times the height of the parent. Content that falls outside of the maximum height is clipped and not accessible. The `expansion_factor` prevents users from having to scroll through extremely long cards that result from unexpectedly large content.

To set the `expansion_factor`, call

```
cardFragment.setExpansionFactor(factor);
```

(where `factor` is a positive float.)

Using `GridViewPager`

This view lets users scroll through a grid of pages and is similar to a `ViewPager` except that it scrolls in both vertically and horizontally. The context stream is an example of a `GridViewPager` where a notification occupies one or more pages of a single row within the grid. Figure 5-4 illustrates a grid of eight pages that form an example `GridViewPager`. Users can only view a single page of this grid at a time, and they can scroll to any adjacent pages.

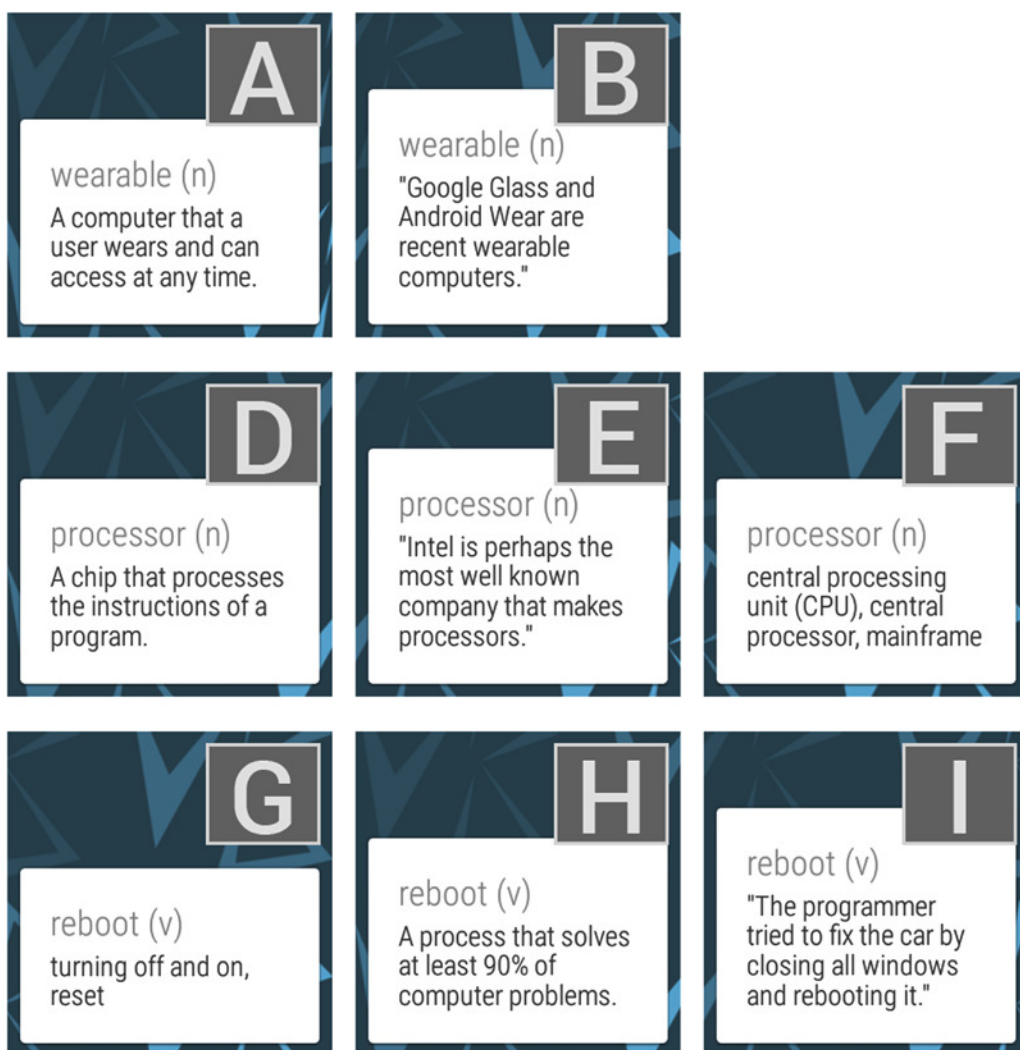


Figure 5-4. An example of a `GridViewPager` with 3 rows and 3 columns. Users can only view a single page of this grid at a time, but they can scroll to any adjacent pages. Note that the word in the top row only has two columns because it does not have synonyms

By default, scrolling up or down from a page always moves back to the first column. For instance, if you're currently viewing page E from Figure 5-4 and you scroll up, you'll land on page A. Similarly, scrolling down from page E leads to page G. This behavior makes sense in the context of Android Wear. Every row of pages usually contains related content that users are expected to browse from left to right. In other words, the pages on the right typically don't make sense without the context provided by the pages to the left. If, for example, you're viewing the weather forecast from Figure 5-1 and you scroll up, it would be confusing if you arrive at another notification's action button.

The content for every page is provided by a subclass of `GridPagerAdapter`. If you want to provide each page as a `Fragment`, extend `FragmentGridPagerAdapter` instead of `GridPagerAdapter`. The `FragmentGridPagerAdapter` class has three required methods you must implement:

```
int getRowCount()
int getColumnCount(int row)
Fragment getItem(int row, int column)
```

The first two indicate the size of content available. Note that `getColumnCount` takes the row number as a parameter. As a result, each row may have a different number of columns. The third method returns the `Fragment` to display at a particular position. By default, a fragment's view is placed on a white background, though you can use a different background by overriding the `getBackgroundForPage` method:

```
Drawable getBackgroundForPage(int row, int column)
```

This method tells Android what image to display behind every page in the `GridViewPager`, and different positions in the `GridViewPager` can use different backgrounds. Alternatively, you could add backgrounds directly to the `Fragment`s, but attempting to scroll past the last card would reveal the white background underneath, as shown in Figure 5-5. Although the difference is subtle, setting the background on the `FragmentGridPagerAdapter` is the better choice.



Figure 5-5. The background of a fragment given to a `GridViewPager` is cut off when users scroll past the last item. To avoid this problem, set the background in the adapter used by `GridViewPager`

Note Background images for Android Wear should have a resolution of 400x400 for fixed backgrounds or 640x400 (or larger) for backgrounds with parallax scrolling. They should be placed in the `res > drawable-nodpi` directory.

Example App #1: Vocab Builder

Now that we've discussed `CardFragment` and `GridViewPager`, we'll see how to implement them in detail. Vocab Builder helps improve your vocabulary by presenting a list of vocabulary words with definitions, example sentences, and synonyms (if available). Figure 5-4 shows an example of a `GridViewPager` used to present three vocabulary words, one per row.

Note The source code for this example is located in the `wear` module of the `WearUiEssentials` project. Start the example with the "Start... Vocab Builder" voice action.

Representing Vocabulary Words

For each vocabulary word, we want to display its definition, category, an example sentence, and an optional list of synonyms. A word's category refers to whether it's a noun, adjective, or adverb. The `VocabularyWord` class stores all this information, and most of its methods are getters and setters. Note that the synonym is not initialized in the constructor since, unlike the rest of the properties, it's optional.

In `VocabularyWord.java`:

```
public class VocabularyWord {
    private String mWord, mExampleSentence, mCategory, mDefinition;
    private String mSynonyms;

    public VocabularyWord(String word, String category, String definition,
        String exampleSentence) {
        mWord = word;
        mCategory = category;
        mDefinition = definition;
        mExampleSentence = exampleSentence;
    }

    public String getWord() {
        return mWord;
    }

    public void setWord(String word) {
        mWord = word;
    }
}
```

```
public String getExampleSentence() {
    return mExampleSentence;
}

public void setExampleSentence(String exampleSentence) {
    mExampleSentence = exampleSentence;
}

public String getCategory() {
    return mCategory;
}

public void setCategory(String category) {
    mCategory = category;
}

public String getDefinition() {
    return mDefinition;
}

public void setDefinition(String definition) {
    mDefinition = definition;
}

public String getSynonyms() {
    return mSynonyms;
}

public void setSynonyms(String synonyms) {
    mSynonyms = synonyms;
}

public boolean hasSynonyms() {
    return mSynonyms != null;
}
}
```

Representing a Vocabulary Word List

A list of vocabulary words is stored as a JSON object within the assets directory. Here is an example of the JSON that represents the vocabulary word list.

```
{
  "words": [
    {
      "word": "wearable",
      "category": "n",
      "definition": "A computer that a user wears and can access at any time.",
      "example_sentence": "Google Glass and Android Wear are recent wearable computers."
    },
  ],
}
```



```

{
    "word": "processor",
    "category": "n",
    "definition": "A chip that processes the instructions of a program.",
    "example_sentence": "Intel is perhaps the most well known company that makes processors.",
    "synonyms": "central processing unit (CPU), central processor, mainframe"
},
...
}

```

In the sample project, we've included a list of nine vocabulary words in the `src > main > assets > sample_vocabulary_list` file. If you are not following along with the sample source code, copy this file into your project's assets folder. To create an assets folder in Android Studio, right click on the wear module in the Android project view and click on `New > Folder > Assets Folder`.

Now that this file is in the assets folder, we'll implement the `VocabularyList` class, which parses the list of vocabulary words from the JSON asset.

In `VocabularyList.java`:

```

public class VocabularyList {
    private List<VocabularyWord> mWords;

    public static VocabularyList fromJson(String jsonVocabularyList) {
        List<VocabularyWord> words = new ArrayList<VocabularyWord>();

        try {
            JSONObject wordListObject = new JSONObject(jsonVocabularyList);
            JSONArray wordListArray = wordListObject.getJSONArray("words");

            for(int i=0; i<wordListArray.length(); ++i) {
                JSONObject wordObject = wordListArray.getJSONObject(i);
                String word = wordObject.getString("word");
                String category = wordObject.getString("category");
                String definition = wordObject.getString("definition");
                String exampleSentence = wordObject.getString("example_sentence");
                String synonyms = wordObject.optString("synonyms");

                VocabularyWord vocabularyWord = new VocabularyWord(word, category,
                    definition, exampleSentence);
                if(!synonyms.isEmpty()) {
                    vocabularyWord.setSynonyms(synonyms);
                }
                words.add(vocabularyWord);
            }

            return new VocabularyList(words);
        } catch (JSONException e) {
            e.printStackTrace();
        }

        return null;
    }
}

```

```

private VocabularyList(List<VocabularyWord> vocabularyWordList) {
    mWords = vocabularyWordList;
}

public int size() {
    return mWords.size();
}

public VocabularyWord getVocabularyWord(int position) {
    return mWords.get(position);
}
}

```

The most important part of the `VocabularyList` is the `fromJson` method, which parses the JSON string into a list of `VocabularyWords`. Now that we're able to retrieve a list of vocabulary words, the next step is to display the content in Android Wear.

Implementing VocabularyAdapter

This class extends `FragmentGridPagerAdapter` and is responsible for creating a fragment and providing content for every cell within the `GridPagerView`. Since every page of a vocabulary word uses a card layout, `VocabularyAdapter` creates a `CardFragment` for each page and populates it with the appropriate content. For each vocabulary word, it will create two or three columns that contain the word's definition, an example sentence, and a list of synonyms (if available), respectively.

1. Declare and initialize member variables.

The `example_bg` drawable contains the image that appears in the background of every page in the `GridPagerView`.

In `VocabularyAdapter.java`:

```

public class VocabularyAdapter extends FragmentGridPagerAdapter {
    private VocabularyList mVocabularyList;
    private Drawable mBackground;

    public VocabularyAdapter(Context context, FragmentManager fm,
        VocabularyList vocabularyList) {
        super(fm);
        mVocabularyList = vocabularyList;
        mBackground = context.getDrawable(R.drawable.example_bg);
    }
    ...
}

```

2. Create a fragment that contains the layout for a given row and column of the `GridViewPager`.

This method returns a `CardFragment` with the appropriate information for a given row and column. `CardFragment.create` is a static method that takes two parameters:

- `CharSequence title`: the title of the card, which is displayed on the first line.
- `CharSequence description`: the text displayed below a card's title.

In this case, the title of a card is "Word (category)", such as "Jump (verb)," and the description of a card is its definition, example sentence, or list of synonyms.

In `VocabularyAdapter.java`:

```
@Override
public Fragment getFragment(int row, int col) {
    VocabularyWord word = mVocabularyList.getVocabularyWord(row);
    String description = null;
    switch (col) {
        case 0:
            description = word.getDefinition();
            break;
        case 1:
            description = "\"" + word.getExampleSentence() + "\"";
            break;
        case 2:
            description = word.getSynonyms();
            break;
    }

    CardFragment cardFragment = CardFragment.create(word.getWord() +
        " (" + word.getCategory() + ")", description);
    cardFragment.setCardGravity(Gravity.BOTTOM);
    cardFragment.setExpansionEnabled(true);
    cardFragment.setExpansionDirection(CardFragment.EXPAND_DOWN);
    cardFragment.setExpansionFactor(2f);

    return cardFragment;
}
```

3. Return the background for a given page.

In this case, we'll use the same background for every page.

In `VocabularyAdapter.java`:

```
@Override
public Drawable getBackgroundForRow(int row) {
    return mBackground;
}
```

4. Return the number of rows and columns.

In this case, there is one row per vocabulary word. Furthermore, a row has three columns if it contains synonyms and two columns otherwise.

In `VocabularyAdapter.java`:

```
@Override
public int getRowCount() {
    return mVocabularyList.size();
}

@Override
public int getColumnCount(int row) {
    if(mVocabularyList.getVocabularyWord(row).hasSynonyms()) {
        return 3;
    } else {
        return 2;
    }
}
```

Before implementing `GridViewPager`, let's see how to implement fixed movement paging for demonstration purposes.

Fixed Movement Paging

As we discussed earlier, scrolling up or down from any page moves back to the first column. This behavior occurs by default but can be overridden. Fixed movement paging refers to an alternate behavior in which scrolling up or down stays on the same column. For instance, with fixed movement paging, scrolling up from Page E in Figure 5-4 lands on page B. To implement fixed movement paging, add the following code to the grid pager's adapter:

```
@Override
public int getCurrentColumnForRow(int row, int currentColumn) {
    return currentColumn;
}
```

The `getCurrentColumnForRow` method tells Android what should be shown above or below the current page. The default method returns 0, which means go back to the first column. This overridden method tells Android to stay on the same column and results in fixed movement paging.

You can test out fixed movement paging by adding the above code to the end of `VocabularyAdapter.java`. Afterwards, be sure to remove or uncomment the method because fixed movement paging is not appropriate for Vocab Builder.

Creating and Populating GridViewPager

All that's left is to create the GridPagerAdapter itself.

1. Create a layout

In res ► layout ► activity_vocabulary.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.GridViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridview"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

2. In the manifest, set the application theme to Theme.DeviceDefault.Light.

In AndroidManifest.xml:

```
<application
    ...
    android:theme="@android:style/Theme.DeviceDefault.Light"
    ...
```

3. Declare VocabularyActivity to be started with the "Vocab Builder" voice action.

This is a standard launcher activity declaration with the exception of taskAffinity, which is only needed because this project uses multiple launcher activities that should be treated independently.

In AndroidManifest.xml:

```
<activity
    android:name=".VocabularyActivity"
    android:label="Vocab Builder"
    android:taskAffinity="com.ocddevelopers.androidwearables.wearuiessentials.VocabTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

4. Implement VocabularyActivity.

In VocabularyActivity.java:

```
public class VocabularyActivity extends Activity {
    private GridViewPager mGridViewPager;
    private String jsonVocabularyList;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_vocabulary);

    VocabularyList vocabularyList = null;

    try {
        // a sample vocabulary list is included in the project as an asset
        InputStream is = getAssets().open("sample_vocabulary_list");

        // read the entire vocabulary list. The delimiter "\A" matches the beginning of
        // input; as a result, the scanner will read the entire stream.
        Scanner scanner = new Scanner(is, "UTF-8").useDelimiter("\\A");
        jsonVocabularyList = scanner.hasNext() ? scanner.next() : "";
        is.close();
        vocabularyList = VocabularyList.fromJson(jsonVocabularyList);
    } catch (IOException e) {
        e.printStackTrace();
    }

    if(vocabularyList == null) {
        throw new RuntimeException("Invalid vocabulary list.");
    }

    mGridViewPager = (GridViewPager)findViewById(R.id.gridview);
    mGridViewPager.setAdapter(new VocabularyAdapter(getFragmentManager(),
        vocabularyList));
}
}
```

Reading the `sample_vocabulary_list` JSON string may seem a little complicated, but that's only because reading a `String` from an `InputStream` is verbose. As the comments explain, this code instantiates a `Scanner` and `split` uses it to split the input string by the "beginning of input" delimiter. Since there is only one beginning of input, the `InputStream` is split into one big chunk.

The JSON string is then passed into the `VocabularyList.fromJson` method, which creates a `VocabularyList` that is used to instantiate the `VocabularyAdapter` used by `GridViewPager`.

Now that we finished implementing `Vocab Builder`, let's learn about more views from the wearable UI library.

Using CircledImageView

This view is essentially an `ImageView` surrounded by a circle of a specified color and size. The easiest way to implement `CircledImageView` is using an XML layout. The following layout, for example, results in Figure 5-6.

```
<android.support.wearable.view.CircledImageView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/circledimageView"
    app:circle_color="#2878ff"
    app:circle_radius="52dip"
    app:circle_radius_pressed="54dip"
    app:circle_border_width="20dip"
    app:circle_border_color="#ff5252"
    android:src="@drawable/ic_full_check"
    android:layout_marginTop="24dip"
    android:layout_gravity="center_horizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

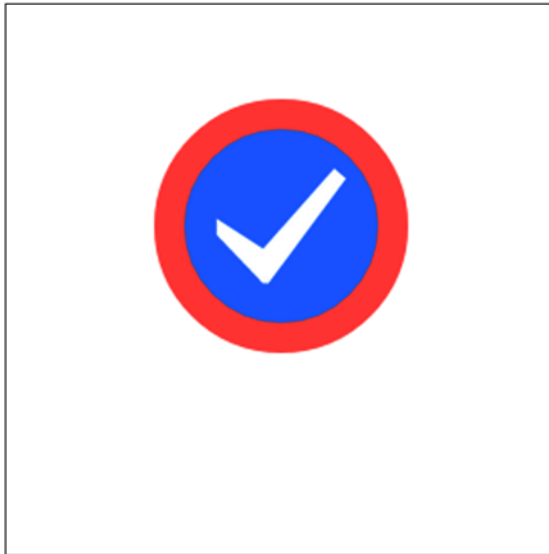


Figure 5-6. A `CircledImageView` displays a check mark icon on a blue circle with a red border

where the `ic_full_check` drawable contains a white checkmark on a transparent background.

The most important properties of `CircledImageView` are `circle_color`, `circle_radius`, `circle_radius_pressed`, `circle_border_width`, and `circle_border_color`. All of these properties are self-explanatory except for `circle_radius_pressed`, which is the radius of the circle that only appears when the user is tapping directly on the `CircledImageView`. Note that these properties should use the `app` namespace instead of the `android` namespace.

Using DelayedConfirmationView

This view subclasses `CircledImageView` and indicates that the user has a few seconds to change or cancel an action. A `DelayedConfirmationView`'s circular border is effectively a progress bar that finishes when it wraps the entire circle. After users set an alarm, for instance, a confirmation screen displays the details of the alarm and gives users a few seconds to change their minds (see Figure 5-7). In this case, the `DelayedConfirmationView` at the bottom-left of the screen indicates the time remaining until the alarm is set. Users can cancel the alarm by swiping from left to right, edit the alarm by tapping the “Edit” button, or set the alarm without delay by tapping the `DelayedConfirmationView`.



Figure 5-7. The native alarm app uses `DelayedConfirmationView` to give the user a few seconds to change or cancel the alarm

`DelayedConfirmationView` can also be implemented with an XML layout.

```
<android.support.wearable.view.DelayedConfirmationView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/delayed_confirmation"
    app:circle_color="@color/blue"
    app:circle_radius="30dip"
    app:circle_radius_pressed="55dip"
    app:circle_border_width="4dip"
    app:circle_border_color="@color/red"
    android:src="@drawable/ic_action_set_alarm"
    android:layout_marginTop="24dip"
    android:layout_gravity="center "
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```


Then, to start the confirmation animation, call

```
mDelayedConfirmationView = (DelayedConfirmationView) findViewById(R.id.delayed_
confirmation);
mDelayedConfirmationView.setTotalTimeMs(5000);
mDelayedConfirmationView.setListener(mDelayedConfirmationListener);
mDelayedConfirmationView.start();
```

where `mDelayedConfirmationListener` is a listener that notifies us if the confirmation progress is complete (with the `onTimerFinished` method) and if the confirmation view was tapped (with the `onTimerSelected` method). Note that `DelayedConfirmationView`'s `setTotalTimeMs` method specifies how much time it takes to call `onTimerFinished`.

```
private DelayedConfirmationView.DelayedConfirmationListener mDelayedConfirmationListener =
    new DelayedConfirmationView.DelayedConfirmationListener() {
        @Override
        public void onTimerFinished(View view) {
            ...
        }

        @Override
        public void onTimerSelected(View view) {
            ...
        }
    };
```

The resulting view is shown in Figure 5-8.

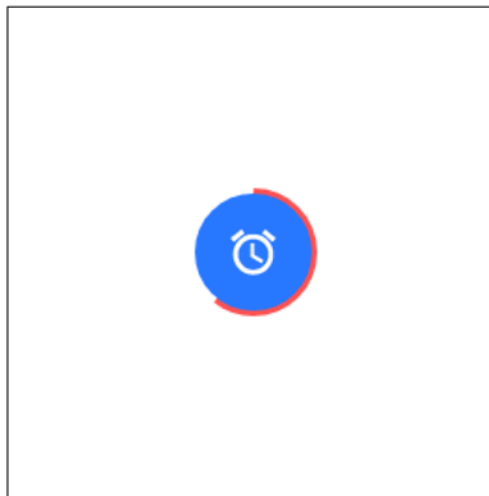


Figure 5-8. A simple example of a `DelayedConfirmationView` with an alarm icon

Using WatchViewStub

Android Wear watches can have a square shape, such as the LG G, or a round shape, such as the Moto 360. `WatchViewStub` helps apps look good on watches of both shapes by inflating different layouts for square and round watches. To use `WatchStubView`, first create separate XML layouts for square and round watches. Then, when creating `WatchStubView`, pass references to the square and round layouts as the `rectLayout` and `roundLayout` attributes, respectively:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.WatchViewStub
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    app:rectLayout="@layout/square_layout"
    app:roundLayout="@layout/round_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

This `WatchViewStub` looks and behaves exactly like the “`square_layout`” and “`round_layout`” layouts on square and round watches, respectively, with one exception. Calling `findViewById` on the `WatchViewStub` always returns null until the layout is inflated. That is, you can’t reference the child views of the square or round layouts directly in your activity’s `onCreate` or in your `Fragment`’s `onCreateView`. `WatchViewStub` will let you know when you can access the child views with the `OnLayoutInflatedListener` listener:

```
WatchViewStub watchViewStub = (WatchViewStub) findViewById(R.id.watchviewstub);
watchViewStub.setOnLayoutInflatedListener(new WatchViewStub.OnLayoutInflatedListener() {
    @Override
    public void onLayoutInflated(WatchViewStub watchViewStub) {
        mExampleTextView = (TextView) watchViewStub.findViewById(R.id.textview);
        ...
    }
}
```

Window Overscan

Watches such as the Moto 360 contain an inset or chin at the bottom of the screen that makes the layout differ from a perfect circle. As a result, its resolution is 320x290 pixels instead of 320x320. Placing a view at the center of the screen with attributes such as `android:layout_centerInParent="true"` or `android:layout_gravity="center"` should center it about (160, 145) since the midpoint of the height is 145. However, this behavior is not usually desirable. An action button for instance, should be centered about (160, 160), which is the center of the screen’s circle. The `window overscan` property of a theme, which is enabled by default, specifies that the root of the view hierarchy is measured as the full size of the circle without taking into account the chin. In the case of the Moto 360, the root of the view hierarchy would be measured as 320x320pixels, which means layouts are centered about (160, 160) as desired.

The only occasion I’ve encountered in which `window overscan` is not desirable is while starting `ConfirmationActivity`, as we’ll see next.

Using ConfirmationActivity

ConfirmationActivity provides visual feedback that shows users whether an action was successful or not. For instance, when users set an alarm, a ConfirmationActivity shows that the operation was successful (see Figure 5-9).

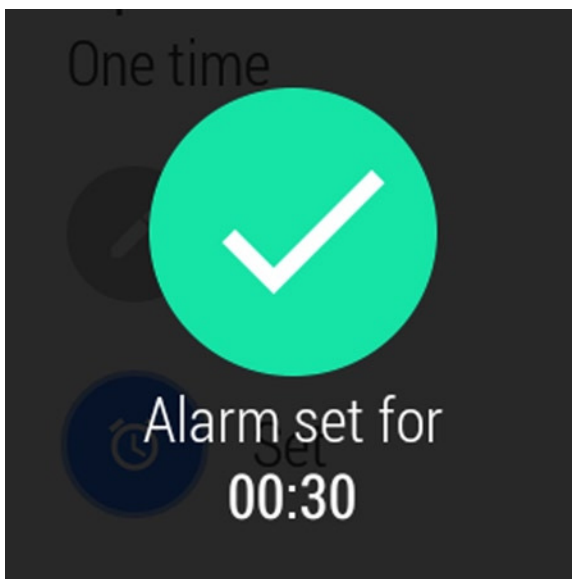


Figure 5-9. An example of ConfirmationActivity's success animation

Using ConfirmationActivity is surprisingly easy. First, declare ConfirmationActivity in AndroidManifest.xml (as we'll see in the following example) and then display the Activity, use the following code.

```
Intent confirmationActivity = new Intent(this, ConfirmationActivity.class)
    .setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_NO_ANIMATION)
    .putExtra(ConfirmationActivity.EXTRA_ANIMATION_TYPE, animationType)
    .putExtra(ConfirmationActivity.EXTRA_MESSAGE, message);
startActivity(confirmationActivity);
```

The intent uses the following flags and extras:

- Flag `Intent.FLAG_ACTIVITY_NEW_TASK`: Isolates ConfirmationActivity in its own task. Since ConfirmationActivity is essentially an animation, it shouldn't affect the current task, which is why we use this flag.
- Flag `Intent.FLAG_ACTIVITY_NO_ANIMATION`: ConfirmationActivity is an animation itself and doesn't need to be animated by the system while entering the screen.

- Extra `ConfirmationActivity.EXTRA_ANIMATION_TYPE`: Tells `ConfirmationActivity` what kind of animation to display. This extra should have a value of `ConfirmationActivity.SUCCESS_ANIMATION`, `ConfirmationActivity.FAILURE_ANIMATION`, or `ConfirmationActivity.OPEN_ON_PHONE_ANIMATION`.
- Extra `ConfirmationActivity.EXTRA_MESSAGE`: Contains a message that will be displayed along with the confirmation.

Additionally, make sure to disable window overscan on `ConfirmationActivity` or else its message may be cut off by the chin at the bottom of the screen of the Moto 360. The following example app demonstrates the appearance of all four animation types and shows how to use many of the views we've covered in this chapter.

Example App #2: Confirmation Demo

This app displays four action buttons, one per page, as shown in Figure 5-10.

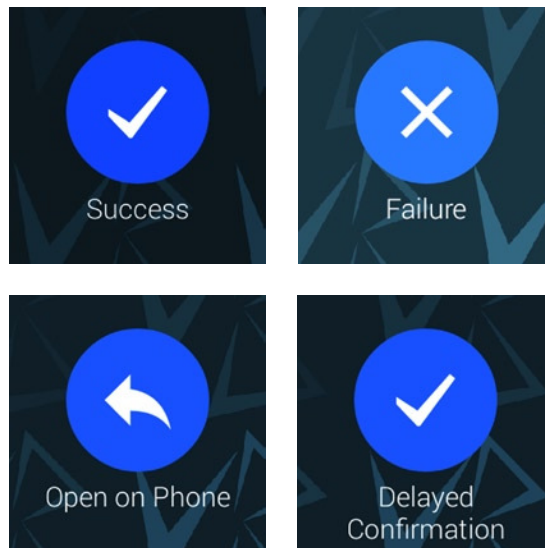


Figure 5-10. The confirmation demo app's four main screens

Tapping on any of the first three buttons shows a `ConfirmationActivity` with the appropriate animation type (see the three left screens of Figure 5-11). Tapping on the last button shows a `DelayedConfirmationView` that triggers a `ConfirmationActivity` after five seconds (see the right screen of Figure 5-11).

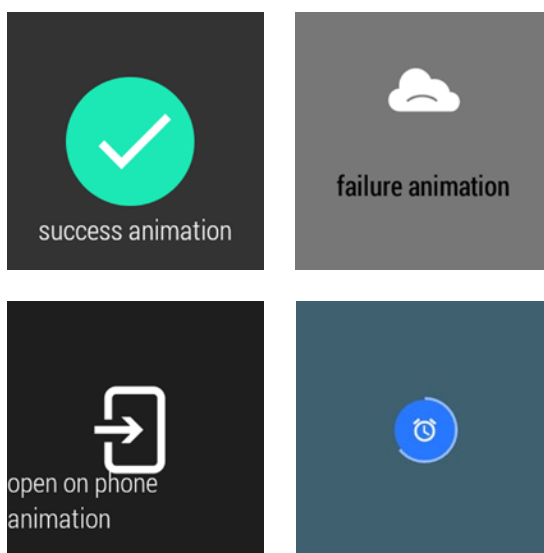


Figure 5-11. The animations demonstrated by the confirmation demo app. From left to right, a `ConfirmationActivity`'s success, failure, and open on phone animations, and a `DelayedConfirmationView`'s progress animation

This app is contrived, but it demonstrates how to use a variety of views from the wearable UI library, including `CircledImageView`, `DelayedConfirmationView`, `WatchviewStub`, and `ConfirmationActivity`.

Note The source code for this example is located in the `wear` module of the `WearUiEssentials` project. Start the example with the “Start... Confirmation Demo” voice action.

Implementing Action Buttons

We use `CircledImageView` to implement action buttons. Buttons in Android Wear typically appear right in the middle of round watches but slightly towards the top of square watches. We'll use `WatchViewStub` to accommodate both cases.

1. Create a layout for square watches.

In `res > layout > demobutton_square.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```

<android.support.wearable.view.CircledImageView
    android:id="@+id/circledimageView"
    app:circle_color="@color/blue"
    app:circle_radius="52dip"
    app:circle_radius_pressed="54dip"
    android:src="@drawable/ic_full_cancel"
    android:layout_marginTop="24dip"
    android:layout_gravity="center_horizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

```

```

<TextView
    android:id="@+id/label"
    android:fontFamily="sans-serif-light"
    android:textSize="20sp"
    android:textColor="@android:color/white"
    android:gravity="center_horizontal"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="4dip"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

```

```
</LinearLayout>
```

2. Create a layout for round watches.

Note that the `CircledImageView` is centered in its parent.

In `res > layout > demobutton_round.xml`:

```

<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

<android.support.wearable.view.CircledImageView
    android:id="@+id/circledimageView"
    app:circle_color="@color/blue"
    app:circle_radius="52dip"
    app:circle_radius_pressed="54dip"
    android:src="@drawable/ic_full_cancel"
    android:layout_centerInParent="true"
    android:layout_marginTop="24dip"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/label"
    android:fontFamily="sans-serif-light"
    android:textSize="20sp"

```

```

    android:textColor="@android:color/white"
    android:gravity="center_horizontal"
    android:layout_below="@id/circledimageview"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="4dip"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

```

```
</RelativeLayout>
```

3. Create a layout that uses `WatchViewStub` to inflate the proper layout, depending on the shape of the watch.

In res ► layout ► demobutton.xml:

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.WatchViewStub
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    app:rectLayout="@layout/demobutton_square"
    app:roundLayout="@layout/demobutton_round"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

```

Creating a Fragment with an Action Button

Now that we have implemented a layout for a button, let's create a `Fragment` that uses this layout and triggers an `Intent` when a button is pressed.

1. Declare member variables.

In `ActionFragment.java`:

```

public class ActionFragment extends Fragment {
    private static final String ARGS_LABEL = "label";
    private static final String ARGS_ICONID = "iconid";
    private static final String ARGS_ACTION_INTENT = "action_intent";
    private CircledImageView mCircledImageView;
    private int mNormalColor, mPressedColor;
    private String mLabel;
    private int mIconId;
    private Intent mActionIntent;
}

```

2. Create a static method that instantiates `ActionFragment`.

An instance of `ActionFragment` requires three parameters:

- `String label`: the label placed below the action button.
- `int iconId`: the ID for the drawable that is placed on the action button.
- `Intent actionIntent`: the intent that is triggered when users tap on the action button.

The newInstance method passes these three parameters to the fragment's arguments. If you are unfamiliar with the newInstance pattern, this stackoverflow thread explains why it's necessary: <http://stackoverflow.com/questions/10450348/do-fragments-really-need-an-empty-constructor>.

In ActionFragment.java:

```
public static ActionFragment newInstance(String label, int iconId, Intent actionIntent) {
    Bundle args = new Bundle();
    args.putString(ARGS_LABEL, label);
    args.putInt(ARGS_ICONID, iconId);
    args.putParcelable(ARGS_ACTION_INTENT, actionIntent);

    ActionFragment fragment = new ActionFragment();
    fragment.setArguments(args);
    return fragment;
}
```

3. In onCreate, initialize variables with the values that were passed into the fragment's arguments in the newInstance method.

In ActionFragment.java:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Bundle args = getArguments();
    if(args == null) {
        throw new RuntimeException("ActionFragment requires arguments.");
    }

    mLabel = args.getString(ARGS_LABEL);
    mIconId = args.getInt(ARGS_ICONID);
    mActionIntent = args.getParcelable(ARGS_ACTION_INTENT);

    Resources res = getResources();
    mNormalColor = res.getColor(R.color.blue);
    mPressedColor = res.getColor(R.color.dark_blue);
}
```

4. Use WatchViewStub to create the layout.

WatchViewStub, in turn, inflates a different layout depending on whether the app is running on a watch with a square or round.

In ActionFragment.java:

```
@Override
public View onCreateView(LayoutInflater inflater,
                        ViewGroup container, Bundle savedInstanceState) {
```



```

/* For round screens, the action button is right in the middle of the screen, but for
 * square screens, the button is slightly above the middle. WatchViewStub lets us
 * provide each shape with a different layout.
 */
WatchViewStub watchViewStub = (WatchViewStub) getActivity().getLayoutInflater().
    inflate(R.layout.demobutton, null);

// we can't access any child views of WatchViewStub until the layout is inflated.
watchViewStub.setOnLayoutInflatedListener(mLayoutInflatedListener);

return watchViewStub;
}

```

5. The child views of WatchViewStub are not accessible until they're inflated. WatchViewStub calls the onLayoutInflated of its OnLayoutInflatedListener listener when the child views are inflated.

In ActionFragment.java:

```

private WatchViewStub.OnLayoutInflatedListener mLayoutInflatedListener =
    new WatchViewStub.OnLayoutInflatedListener() {
    @Override
    public void onLayoutInflated(WatchViewStub watchViewStub) {
        TextView label = (TextView)watchViewStub.findViewById(R.id.label);
        label.setText(mLabel);

        mCircledImageView =
            (CircledImageView)watchViewStub.findViewById(R.id.circledimageview);
        mCircledImageView.setImageResource(mIconId);

        // Give the button a darker color (mPressedColor) when the user touches any
        // part of the screen. When the user removes her finger from the screen,
        // go back to the original color (mNormalColor) and start the action intent.
        watchViewStub.setOnTouchListener(mTouchListener);
    }
};

```

6. Buttons need to display touch feedback when they are tapped. Change its color when a user is tapping on the button.

In ActionFragment.java:

```

private View.OnTouchListener mTouchListener = new View.OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        switch (event.getActionMasked()) {
            case MotionEvent.ACTION_DOWN:
                mCircledImageView.setCircleColor(mPressedColor);
                return true;
            case MotionEvent.ACTION_MOVE:
                return true;
        }
    }
};

```

```

        case MotionEvent.ACTION_UP:
            startActivity(mActionIntent);
            mCircledImageView.setCircleColor(mNormalColor);
            return true;
        case MotionEvent.ACTION_CANCEL:
            mCircledImageView.setCircleColor(mNormalColor);
            return true;
        default:
            return false;
    }
}
};

```

A `MotionEvent.ACTION_CANCEL` action is triggered when the current gesture has been aborted, in which case `MotionEvent.ACTION_UP` will not be triggered. A gesture can get aborted if, for instance, your finger swipes outside of the screen. Handling `ACTION_CANCEL` ensures that the circle always returns to the original blue color when the user isn't touching the screen.

Implementing DelayedConfirmationActivity

In this section, we'll create an Activity that displays a `DelayedConfirmationView` that lasts for five seconds, after which it starts `ConfirmationActivity`. If the user taps on the `DelayedConfirmationView` before the five seconds are over, the `ConfirmationActivity` is immediately displayed. Tapping on a confirmation view typically tells the app to execute the action without delay.

1. Create a layout.

This layout contains a `DelayedConfirmationView` with an alarm icon on top of a blue circle.

In `res > layout > activity_delayed_confirmation.xml`:

```

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:background="#3e606f"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.wearable.view.DelayedConfirmationView
        android:id="@+id/delayed_confirmation"
        app:circle_color="@color/blue"
        app:circle_radius="30dip"
        app:circle_radius_pressed="55dip"
        app:circle_padding="4dip"
        app:circle_border_width="4dip"
        app:circle_border_color="#ff94bcff"
        android:src="@drawable/ic_action_set_alarm"
        android:layout_centerInParent="true"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</RelativeLayout>

```

2. Declare the activity.

In `AndroidManifest.xml`:

```
<activity
    android:name=".DelayedConfirmationActivity"
    android:label="Delayed Confirmation Demo" >
</activity>
```

3. Declare and initialize `DelayedConfirmationView`.

In `DelayedConfirmationActivity.java`:

```
public class DelayedConfirmationActivity extends Activity {
    private DelayedConfirmationView mDelayedConfirmationView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_delayed_confirmation);

        mDelayedConfirmationView = (DelayedConfirmationView) findViewById(R.id.delayed_
            confirmation);
        mDelayedConfirmationView.setTotalTimeMs(5000);

        mDelayedConfirmationView.setListener(mDelayedConfirmationListener);
        mDelayedConfirmationView.start();
    }
    ...
}
```

4. Start `ConfirmationActivity` when `DelayedConfirmationView` is completed (that is, in `onTimerFinished`) or when the user taps on the `DelayedConfirmationView` (that is, on `onTimerSelected`).

In `DelayedConfirmationActivity.java`:

```
private DelayedConfirmationView.DelayedConfirmationListener mDelayedConfirmationListener =
    new DelayedConfirmationView.DelayedConfirmationListener() {
    @Override
    public void onTimerFinished(View view) {
        startConfirmationActivity(ConfirmationActivity.SUCCESS_ANIMATION,
            "Alarm set for 08:30");
        finishActivity();
    }

    @Override
    public void onTimerSelected(View view) {
        startConfirmationActivity(ConfirmationActivity.SUCCESS_ANIMATION,
            "Alarm set for 08:30");
        finishActivity();
    }
};
```

```
private void finishActivity() {
    // stop the delayed confirmation by settings its total time to zero, at least until
    // a better API is available.
    mDelayedConfirmationView.setTotalTimeMs(0);
    finish();
}
```

5. Implement startConfirmationActivity.

This method starts a ConfirmationActivity with the animation type and message specified by the parameters.

In DelayedConfirmationActivity.java:

```
private void startConfirmationActivity(int animationType, String message) {
    Intent confirmationActivity = new Intent(this, ConfirmationActivity.class)
        .setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_NO_ANIMATION)
        .putExtra(ConfirmationActivity.EXTRA_ANIMATION_TYPE, animationType)
        .putExtra(ConfirmationActivity.EXTRA_MESSAGE, message);
    startActivity(confirmationActivity);
}
```

Implementing ConfirmationDemoActivity

The main Activity of this app is based on a GridViewPager with a single row and four columns, each of which contains an ActionFragment.

1. Create a layout.

The layout only contains a GridViewPager.

In res ► layout ► activity_confirmation_demo.xml:

```
<android.support.wearable.view.GridViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridviewpager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

2. Create a style for a theme that disables window overscan.

In res ► values ► styles.xml:

```
<style name="NoOverscan" parent="android:Theme.DeviceDefault.Light">
    <item name="android:windowOverscan">false</item>
</style>
```

3. Declare this activity and ConfirmationActivity.

ConfirmationDemoActivity uses the declaration of a standard launcher activity with the exception of taskAffinity, which is only needed because this project uses multiple launcher activities that should be treated independently.

Even though ConfirmationActivity is part of the wearable UI library, apps still need to declare it in the manifest. Also, declare it with the NoOverscan theme to ensure it the message isn't cutoff on the Moto 360.

In `AndroidManifest.xml`:

```
<activity
    android:name=".ConfirmationDemoActivity"
    android:label="Confirmation Demo"
    android:taskAffinity="com.ocddevelopers.androidwearables.wearuiessentials.
ConfirmationTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

<activity
    android:name="android.support.wearable.activity.ConfirmationActivity"
    android:theme="@style/NoOverscan" />
```

4. Declare member variables.

In `ConfirmationDemoActivity.java`:

```
public class ConfirmationDemoActivity extends Activity {
    private Fragment[] mFragments;
    private GridViewPager mGridViewPager;
    ...
}
```

5. In `onCreate`, create a list of `ActionFragments` which contain action buttons that start `ConfirmationActivity` with different animations when pressed.

The `makeConfirmationActivityIntent` method and the `ConfirmationDemoGridPagerAdapter` class are implemented in the next two steps.

In `ConfirmationDemoActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_confirmation_demo);

    // The first three fragments demonstrate ConfirmationActivity's success, failure, and
    // open on phone animations, respectively. The fourth fragment demonstrates the use of
    // DelayedConfirmationView.
    mFragments = new Fragment[] {
        ActionFragment.newInstance("Success", R.drawable.ic_full_check,
            makeConfirmationActivityIntent(ConfirmationActivity.SUCCESS_ANIMATION,
                "success animation")),
        ActionFragment.newInstance("Failure", R.drawable.ic_full_cancel,
            makeConfirmationActivityIntent(ConfirmationActivity.FAILURE_ANIMATION,
                "failure animation")),
        ActionFragment.newInstance("Open on Phone", R.drawable.ic_full_reply,
            makeConfirmationActivityIntent(ConfirmationActivity.OPEN_ON_PHONE_
                ANIMATION,
                "open on phone animation")),
    };
```

```

        ActionFragment.newInstance("Delayed Confirmation", R.drawable.ic_full_check,
            new Intent(this, DelayedConfirmationActivity.class))
    };

    mGridViewPager = (GridViewPager) findViewById(R.id.gridviewpager);
    mGridViewPager.setAdapter(new ConfirmationDemoGridPagerAdapter(this,
        getFragmentManager(), mFragments));
}

```

6. Implement makeConfirmationActivityIntent.

This method creates an Intent that starts ConfirmationActivity with the animation a message specified by the parameters.

In ConfirmationDemoActivity.java:

```

private Intent makeConfirmationActivityIntent(int animationType, String message) {
    Intent confirmationActivity = new Intent(this, ConfirmationActivity.class)
        .setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_NO_ANIMATION)
        .putExtra(ConfirmationActivity.EXTRA_ANIMATION_TYPE, animationType)
        .putExtra(ConfirmationActivity.EXTRA_MESSAGE, message);
    return confirmationActivity;
}

```

7. Create an inner class that extends FragmentGridPagerAdapter and provides an interface for the GridViewPager to access the fragments created in onCreate.

We learned how to extend FragmentGridPagerAdapter in the Vocab Builder example. Note that, in this step, the getRowCount method returns 1 since the grid only contains pages across a single row.

In ConfirmationDemoActivity.java:

```

public static class ConfirmationDemoGridPagerAdapter extends FragmentGridPagerAdapter {
    private Drawable mBackground;
    private Fragment[] mDemoFragments;

    public ConfirmationDemoGridPagerAdapter(Context context, FragmentManager fm,
        Fragment[] fragments) {
        super(fm);
        mBackground = context.getDrawable(R.drawable.definition_bg);
        mDemoFragments = fragments;
    }

    @Override
    public Fragment getFragment(int row, int column) {
        return mDemoFragments[column];
    }

    @Override
    public int getRowCount() {
        return 1;
    }
}

```

```

@Override
public int getColumnCount(int row) {
    return mDemoFragments.length;
}

@Override
public Drawable getBackgroundForPage(int row, int column) {
    return mBackground;
}
}

```

At this point, you should be able to run the example app and see all the different types of confirmation animations.

Before implementing the next example app, we must learn about `BoxInsetLayout`.

Using `BoxInsetLayout`

This layout extends `FrameLayout` and behaves differently on a square watch than on a round watch:

- on a square watch, `BoxInsetLayout` has no effect whatsoever.
- on a round watch, `BoxInsetLayout` positions one or more edges of a child view inside the square that is inscribed within the watch.

The `layout_box` parameter, which is placed in a child view's XML, specifies what edges should be contained within the box. Figure 5-12 illustrates how several values of `layout_box` result in different positions of a child view.

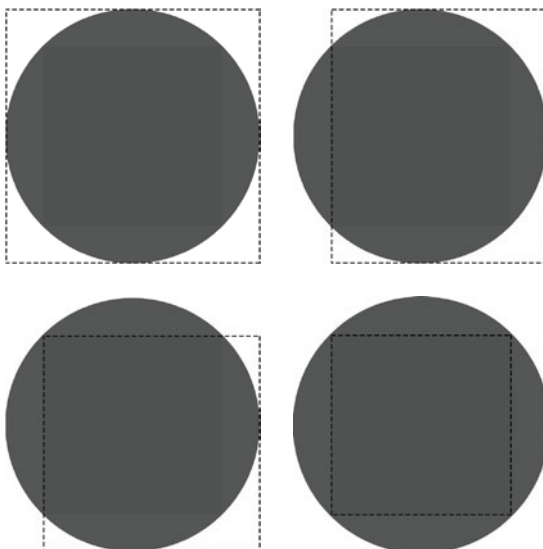


Figure 5-12. A view represented by the dashed line is contained inside `BoxInsetLayout`. The view's position depends on the value of `layout_box`. From left to right: `layout_box` is not specified, `layout_box` is "left", `layout_box` is "lefttop", and `layout_box` is "all"

BoxInsetLayout lets you make a single layout that looks good in both square and round watches.

The next example app combines many of the skills we've learned so far, including notifications. Additionally, it demonstrates how to create custom notification layouts.

Example App #3: Running Stats

This app demonstrates how to display basic stats for a person who is running: duration, distance, speed, and calories burned. For the purposes of this example, we'll display simulated data since this chapter does not cover location or GPS. The app's layout is shown in Figure 5-13.

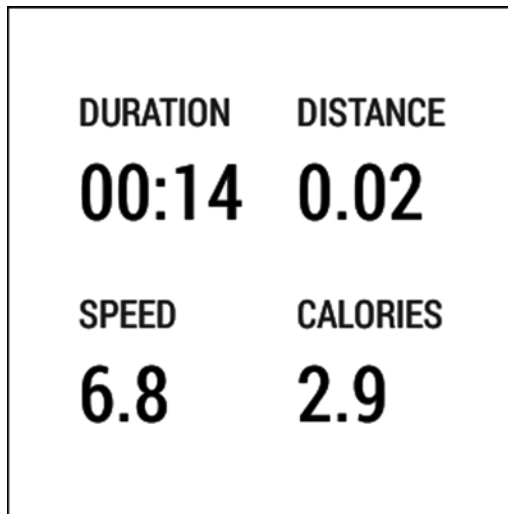


Figure 5-13. Displaying the basic running stats

Note The source code for this example is located in the wear module of the WearUiEssentials project. Start the example with the “Start... Run Stats” voice action.

Simulating Running Stats

Since this chapter focuses on building user interfaces for Android Wear, this example does not deal with location or GPS and, instead, displays simulated stats for demonstration purposes. The `SimulatedRunStatsReceiver` class generates the simulated stats based on an arbitrarily selected speed. That is, this class assumes the user is moving at a certain speed and uses this value to calculate distance travelled and calories burned. In a real implementation, the speed and distance would be measured with GPS.

It's not essential to understand the implementation of this class because it does not contain any functionality that is specific to Android Wear. As long as you understand that this class generates simulated values, you could skip the rest of this section if desired.

1. Declare constants and member variables.

The simulated speed of the runner oscillates between `MIN_SPEED_MPH` and `MAX_SPEED_MPH` with a period of `PERIOD_IN_SECONDS`. The rest of the constants are simply used for unit conversions.

In `SimulatedRunStatsReceiver.java`:

```
public class SimulatedRunStatsReceiver {
    private static final float MILLIS_PER_SECOND = 1000f;
    private static final float SECONDS_PER_MINUTE = 60;
    private static final float SECONDS_PER_HOUR = 3600;
    private static final float PERIOD_IN_SECONDS = 8*SECONDS_PER_MINUTE;
    private static final float MPH_PER_METER_PER_SECOND = 2.23694f;
    private static final float METERS_PER_MILE = 1609.34f;
    private static final int MIN_SPEED_MPH = 5;
    private static final int MAX_SPEED_MPH = 8;
    private RunCallback mRunCallback;
    private Handler mHandler;
    private float mSimulatedDistance, mSimulatedCalories;
    private long mStartTimeMillis, mPrevTimeMillis;
    ...
}
```

2. Implement the `RunCallback` inner interface, which is used to send an observer run stats any time they're updated.

In `SimulatedRunStatsReceiver.java`:

```
public interface RunCallback {
    void runStatsReceived(float distance, float speed, float calories);
}
```

3. Initialize the handler in the constructor and provide setters and getters for an instance of `RunCallback`.

In `SimulatedRunStatsReceiver.java`:

```
public SimulatedRunStatsReceiver() {
    mHandler = new Handler();
}

public RunCallback getRunCallback() {
    return mRunCallback;
}
```

```
public void setRunCallback(RunCallback runCallback) {
    mRunCallback = runCallback;
}
```

4. Implement the `startRun` and `stopRun` methods, which tell the class to start or stop generating new values, respectively.

In `SimulatedRunStatsReceiver.java`:

```
public void startRun() {
    mSimulatedDistance = 0f;
    mSimulatedCalories = 0f;
    mStartTimeMillis = SystemClock.elapsedRealtime();
    mPrevTimeMillis = mStartTimeMillis;
    mStatsRunnable.run();
}

public void stopRun() {
    mHandler.removeCallbacks(mStatsRunnable);
}
```

5. Implement a `Runnable` that is repeatedly called by the handler and generates new stats.

The calculations in this step generate coherent values that make sense as a whole. The user's speed is set to a value that oscillates between `MIN_SPEED_MPH` and `MAX_SPEED_MPH`. The formula that generates the speed uses a sin function to generate the speed. There is nothing significant about this formula or the values chosen for the min/max speed. Based on the speed, the rest of this step calculates the distance travelled and calories burned.

In `SimulatedRunStatsReceiver.java`:

```
private Runnable mStatsRunnable = new Runnable() {
    @Override
    public void run() {
        /* All simulated values should be coherent. We will choose a reasonable running
           speed
           use it to calculate distance and calories. */
        long now = SystemClock.elapsedRealtime();
        float timeElapsedSeconds = (now - mStartTimeMillis)/MILLIS_PER_SECOND;
        float deltaTimeSeconds = (now - mPrevTimeMillis)/MILLIS_PER_SECOND;

        /* Speed is chosen to fluctuate between MIN_SPEED_MPH and MAX_SPEED_MPH. We
           accomplish
           this with with a sinusoid of period PERIOD_IN_SECONDS.

           Don't spend too much time trying to figure out how this function works. It's not
           important. Just understand that the speed is generated as a function of time. */
        float simulatedSpeedMph = (float) (0.5*MIN_SPEED_MPH + 0.5*MAX_SPEED_MPH +
            0.5*(MAX_SPEED_MPH - MIN_SPEED_MPH)*
            Math.sin(2 * Math.PI / PERIOD_IN_SECONDS * timeElapsedSeconds));
```

```

// The user travelled for a time of deltaTimeSeconds at a speed of
    mSimulatedSpeedMph.
// Calculate how much distance the user travelled and add it to the total distance.
double simulatedSpeedMetersPerSecond = simulatedSpeedMph / MPH_PER_METER_PER_SECOND;
mSimulatedDistance += deltaTimeSeconds*simulatedSpeedMetersPerSecond;

// Calculate how many calories the user burned in th time interval. We'll do so
    with the
// formulas provided by the ACSM at http://certification.acsm.org/metabolic-calcs
double simulatedSpeedMetersPerMin = simulatedSpeedMetersPerSecond*SECONDS_PER_
MINUTE;

/* grade is another way to represent the incline of the run. For more detail, see
    http://www.livestrong.com/article/422012-what-is-10-degrees-in-incline-on-a-treadmill/
    note that a grade of 0.01 (that is, 1%) approximates running outside.*/
double grade = 0.01;
// vo2 is the s oxygen consumption
double vo2 = 3.5 + 0.2*simulatedSpeedMetersPerMin+ 0.9*simulatedSpeedMetersPerMin
*grade;
// For more info on metabolic rate: http://en.wikipedia.org/wiki/Metabolic\_
    equivalent
double metabolicRate = vo2 / 3.5;

// In a real app, the weight would be inserted by the user. We use an arbitrary value
// for demonstration purposes.
double weightInKg = 70;
double deltaTimeInHours = deltaTimeSeconds/SECONDS_PER_HOUR;
double deltaCalories = metabolicRate*weightInKg*deltaTimeInHours;
mSimulatedCalories += deltaCalories;

if(mRunCallback != null) {
    float distanceInMiles = mSimulatedDistance / METERS_PER_MILE;
    mRunCallback.runStatsReceived(distanceInMiles, simulatedSpeedMph,
        mSimulatedCalories);
}

mHandler.postDelayed(mStatsRunnable, 1000);
mPrevTimeMillis = now;
}
};

```

Now that we can generate run stats, let's implement an activity that displays these values on Android Wear.

Implementing RunStatsActivity

This activity uses the `SimulatedRunStatsReceiver` class to generate simulated run stats and display them on the screen.

1. Create a layout.

This layout uses `BoxInsetLayout` to make sure that all of the content stays within the screen's bounds, regardless of whether it's on a square or a round watch. The `GridLayout` specifies a `layout_box` of "left", which pushes its content to the right of a round watch. After implementing this example, I recommend deleting the `layout_box` line and running the code once again to see how it affects the layout on a round watch. Removing this line will have no effect on a square watch. Note that we do not need to use a `layout_box` of "top" since we are centering the entire layout vertically.

In res ► layout ► activity_run_stats.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <GridLayout
        app:layout_box="left"
        android:paddingLeft="8dip"
        android:layout_gravity="center_vertical"
        android:columnCount="2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">

        <TextView
            android:text="Duration"
            style="@style/StatTitle" />

        <TextView
            android:text="Distance"
            style="@style/StatTitle" />

        <TextView
            android:id="@+id/duration"
            android:text="00:00"
            android:layout_marginRight="16dip"
            style="@style/StatValue" />

        <TextView
            android:id="@+id/distance"
            android:text="0.0"
            style="@style/StatValue" />
    </GridLayout>
</android.support.wearable.view.BoxInsetLayout>
```

```

    <TextView
        android:text="Speed"
        style="@style/StatTitle"
        android:layout_marginTop="16dip" />

    <TextView
        android:text="Calories"
        style="@style/StatTitle"
        android:layout_marginTop="16dip" />

    <TextView
        android:id="@+id/speed"
        android:text="0.0"
        style="@style/StatValue" />

    <TextView
        android:id="@+id/calories"
        android:text="0.0"
        style="@style/StatValue" />

</GridLayout>
</android.support.wearable.view.BoxInsetLayout>

```

2. Declare `RunStatsActivity` to be started with the “Run Stats” voice action.

In `AndroidManifest.xml`:

```

<activity
    android:name=".RunStatsActivity"
    android:label="Run Stats">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

```

3. Declare and initialize member variables.

In `RunStatsActivity.java`:

```

public class RunStatsActivity extends Activity {
    private SimulatedRunStatsReceiver mStatsReceiver;
    private TextView mDuration, mDistance, mSpeed, mCalories;
    private Handler mHandler;
    private long mStartTime;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_run_stats);
    }
}

```

```

mDuration = (TextView) findViewById(R.id.duration);
mDistance = (TextView) findViewById(R.id.distance);
mSpeed = (TextView) findViewById(R.id.speed);
mCalories = (TextView) findViewById(R.id.calories);

mStatsReceiver = new SimulatedRunStatsReceiver();
mStatsReceiver.setRunCallback(mRunCallback);
mStartTime = SystemClock.elapsedRealtime();
mStatsReceiver.startRun();

mHandler = new Handler();
mHandler.postDelayed(mDurationRunnable, 250);
}
...

```

4. Receive updated run stats.

When the generated stats are updated, the `SimulatedRunStatsReceiver` instance calls `RunCallback`'s `runStatsReceived` method. Display these values in the interface.

In `RunStatsActivity.java`:

```

private SimulatedRunStatsReceiver.RunCallback mRunCallback = new SimulatedRunStatsReceiver.
RunCallback() {
    @Override
    public void runStatsReceived(float distance, float speed, float calories) {
        mDistance.setText(String.format(Locale.US, "%.2f", distance));
        mSpeed.setText(String.format(Locale.US, "%.1f", speed));
        mCalories.setText(String.format(Locale.US, "%.1f", calories));
    }
};

```

5. Periodically update the duration.

The `runStatsReceived` method from step 3 does not update the duration. To update the duration, calculate the time elapsed since the activity was started and display it with a `HH:MM:SS` format.

In `RunStatsActivity.java`:

```

private Runnable mDurationRunnable = new Runnable() {
    @Override
    public void run() {
        long durationInSeconds = (SystemClock.elapsedRealtime() - mStartTime + 500L)/1000L;
        int hours = (int) (durationInSeconds/3600);
        int minutes = (int) ((durationInSeconds%3600)/60);
        int seconds = (int) (durationInSeconds%60);
        String time = String.format("%02d:02d:%02d", hours, minutes, seconds);

        if(!time.equals(mDuration.getText())) {
            mDuration.setText(time);
        }
    }
};

```

```

        mHandler.postDelayed(mDurationRunnable, 250);
    }
};

```

6. When the activity is destroyed, stop generating new stats.

In `RunStatsActivity.java`:

```

@Override
protected void onDestroy() {
    mStatsReceiver.stopRun();
    super.onDestroy();
}

```

The next example app demonstrates how to use `RunStatsActivity` to display run stats in a custom notification.

Example App #4: Creating a Custom Notification

Displaying running stats in an activity may not be such a great idea since it would occupy the watch's foreground throughout a user's entire run. A user wouldn't even be able to check the time while running the app. Instead, we'll display the running stats in a custom notification that takes up the entire screen (see Figure 5-14).

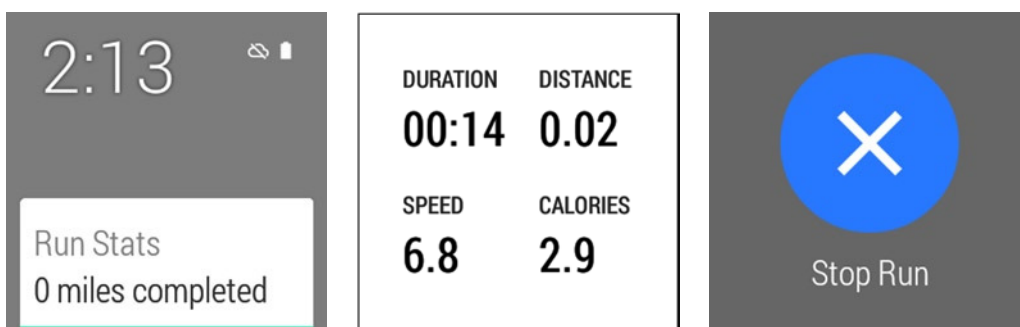


Figure 5-14. Displaying running stats in a custom notification. The unexpanded notification only shows its content title and text (left). The expanded notification fills the entire screen (middle). An action allows the user to stop the run and remove the notification (right)

Note The source code for this example is located in the `wear` module of the `WearUiEssentials` project. Start the example with the “Start... Run Stats Notification” voice action.

Creating a Custom Notification

The `RunStatsNotificationUtil` class contains helper methods to create and remove the custom notification.

1. Create an activity that contains the custom layout and behavior you want your notification to have.

This example's custom notification uses the `RunStatsActivity` class we implemented in the previous section.

2. Modify the declaration of `RunStatsActivity` and make it exported, embeddable, and give it an empty `taskAffinity`.

Activities that are embedded into custom notifications require these parameters.

In `AndroidManifest.xml`:

```
<activity
    android:name=".RunStatsActivity"
    android:allowEmbedded="true"
    android:exported="true"
    android:label="Run Stats"
    android:taskAffinity="" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3. Declare constants.

The only constant in this class represents the ID of the custom notification.

In `RunStatsNotificationUtil.java`:

```
public class RunStatsNotificationUtil {
    public static final int NOTIFICATION_ID = 1;
    ...
}
```

4. Create a `PendingIntent` that starts an activity with the desired layout.

In `RunStatsNotificationUtil.java`:

```
private static PendingIntent makeDisplayPendingIntent(Context context) {
    Intent statsActivityIntent = new Intent(context, RunStatsActivity.class);
    return PendingIntent.getActivity(context, 0,
        statsActivityIntent, PendingIntent.FLAG_CANCEL_CURRENT);
}
```


5. Create a notification.

Use the `PendingIntent` from the previous step as the notification's display intent. Additionally, add an action that cancels the notification. This action should trigger a `PendingIntent` obtained from `RunStatsNotificationService`'s `makeStopIntent` method, which we'll implement in the next section.

Note the following characteristics of the notification:

- We call `WearableExtender`'s `setCustomSizePreset` method with the `SIZE_FULL_SCREEN` parameter to specify that the notification should occupy the entire screen.
- Even though the small icon of this notification is not displayed in the custom notification, this argument is still required merely because it's required of all notifications.
- The notification is ongoing and can only be cancelled with the "Stop Run" action.

In `RunStatsNotificationUtil.java`:

```
public static void showNotification(Context context) {
    PendingIntent statsActivityPendingIntent = makeDisplayPendingIntent(context);

    Intent stopRunIntent = RunStatsNotificationService.makeStopIntent(context);
    PendingIntent stopRunPendingIntent = PendingIntent.getService(context, 0,
        stopRunIntent, PendingIntent.FLAG_CANCEL_CURRENT);

    Notification runStatsNotification = new NotificationCompat.Builder(context)
        .setContentTitle("Run Stats")
        .setContentText("")
        .setSmallIcon(R.drawable.ic_launcher)
        .setPriority(Notification.PRIORITY_HIGH)
        .setOngoing(true)
        .addAction(R.drawable.ic_full_cancel, "Stop Run", stopRunPendingIntent)
        .extend(new NotificationCompat.WearableExtender()
            .setDisplayIntent(statsActivityPendingIntent)
            .setCustomSizePreset(NotificationCompat.WearableExtender.SIZE_FULL_SCREEN))
        .build();

    NotificationManagerCompat notificationManager = NotificationManagerCompat.from(context);
    notificationManager.notify(NOTIFICATION_ID, runStatsNotification);
}
```

6. Create a method that cancels the notification.

In `RunStatsNotificationUtil.java`:

```
public static void hideNotification(Context context) {
    NotificationManagerCompat notificationManager = NotificationManagerCompat.from(context);
    notificationManager.cancel(NOTIFICATION_ID);
}
```

Stopping the Custom Notification

When a user clicks on the “Stop Run” action of the run stats notification, it starts a service that removes the notification from the context stream.

1. Declare the service.

In `AndroidManifest.xml`:

```
<service
    android:name=".RunStatsNotificationService"
    android:exported="false" >
</service>
```

2. Implement `RunStatsNotificationService`.

This class contains a helper method to create an intent that removes the notification when started. When this intent is started, Android calls `RunStatsNotificationService`'s `onHandleIntent` with the `ACTION_STOP_RUN` action.

In `RunStatsNotificationService.java`:

```
public class RunStatsNotificationService extends IntentService {
    private static final String ACTION_STOP_RUN =
        "com.ocddevelopers.androidwearables.wearuiessentials.action.STOP_RUN";

    public static Intent makeStopIntent(Context context) {
        Intent intent = new Intent(context, RunStatsNotificationService.class);
        intent.setAction(ACTION_STOP_RUN);
        return intent;
    }

    public RunStatsNotificationService() {
        super("RunStatsNotificationService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        if (intent != null) {
            final String action = intent.getAction();
            if (ACTION_STOP_RUN.equals(action)) {
                RunStatsNotificationUtil.hideNotification(this);
            }
        }
    }
}
```

Starting the Custom Notification

The last step is to start the notification from a voice action. We do this by creating the notification from an activity and immediately finishing the activity so it never becomes visible.

1. Declare `RunStatsNotificationActivity` to be started with the “Run Stats Notification” voice action.

This is a standard launcher activity declaration with the exception of `taskAffinity`, which is only needed because this project uses multiple launcher activities that should be treated independently.

In `AndroidManifest.xml`:

```
<activity
    android:name=".RunStatsNotificationActivity"
    android:label="Run Stats Notification"
    android:taskAffinity="com.ocddevelopers.androidwearables.wearuiessentials.RunTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

2. Implement `RunStatsNotificationActivity`.

In `RunStatsNotificationActivity.java`:

```
public class RunStatsNotificationActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_run_stats_notification);

        RunStatsNotificationUtil.showNotification(this);

        finish();
    }
}
```

Run the example to see the custom notification in action.

Summary

We learned how to use many views and widgets from the wearable UI library and created several example apps: one for studying vocabulary, another for demoing confirmations, and another for displaying running stats. We also used the activity that displays running stats to create a custom notification that takes the entire screen. In the next chapter, we’ll see how Android Wear can communicate with a paired handheld device, which is one of the most powerful features of Android Wear in my opinion.

The Wearable Data Layer API

While Android Wear is great for simple and timely interactions, more complex and lengthy interactions should be done on a handheld device. For example, viewing an email you just received is useful and convenient but searching your inbox for a particular message is complex enough that it should only be done from a handheld. Additionally, capabilities that require an elevated amount of battery or processing should be performed on the handheld. For instance, Android Wear's navigation app receives location updates that are measured on the handheld's GPS when possible.

In this chapter, we'll learn about the wearable data layer API, which lets a handheld device communicate and share data with a paired Android Wear device. We'll begin by seeing how to connect to the wearable data layer and then we'll learn about the Node, Message, and Data APIs.

The Wearable Data Layer

The wearable data layer lets a handheld and a paired Android Wear device communicate with each other and share data over Bluetooth. A device uses the wearable data layer API to send data or messages that are tagged with a unique identifier called a path. The data layer then transmits the message to the other device and notifies the receiving device as it receives new data. The receiver uses the path to determine what data has been updated. Thus, the wearable data layer API provides public APIs that allow developers to share data between devices without having to deal with all the nuances of communication and synchronization, such as data serialization and error handling. At a high level, the wearable data layer provides two types of communication:

- the *Data API*, which shares and synchronizes data between both devices, and
- the *Message API*, which is a one-way communication mechanism that sends messages from one device to another and is good for remote procedure calls.

Additionally, the wearable data layer contains the Node API, which lets apps determine if the wearable data layer is connected to another device. We'll discuss the details of all these APIs shortly, but regardless of which API we use, the first step is to connect to the `GoogleApiClient` class.

Connecting to `GoogleApiClient`

Google Play Services provides new APIs to developers on a regular basis without having to wait for updated versions of Android. Google Play Services is not part of the Android platform, but an app that Google Play can update automatically just like any other app. These updates can contain new APIs that are compatible as far back as Gingerbread (that is, API 10).

The wearable data layer API is part of Google Play Services and is managed by the `GoogleApiClient` class. In general, `GoogleApiClient` manages a network connection between a device and a service and is not limited to the data layer API—it can also be used to connect to services such as Google+ or Google Drive.

Example #1: Establishing an Asynchronous Connection

Since connecting to Google Play Services is a lengthy operation, trying to connect synchronously on the UI thread would lead to an unresponsive app. Our code can avoid monopolizing the UI thread by connecting to Google Play Services asynchronously.

Note Accessing network or file resources often takes a relatively long time. A synchronous method blocks the execution of code until it obtains a result. An asynchronous method, on the other hand, immediately returns even if its result is not yet available. The result is then calculated, perhaps on another thread, while the code continues to execute. The caller is then notified when the result becomes available.

You must connect to Google Play Services regardless of whether you intend to send or receive data, and regardless of whether you're on a handheld or Android Wear device. The code is the same for all cases. Although this example demonstrates how to connect to `GoogleApiClient` from a handheld device, the process would be identical from an Android Wear device.

Note The source code for this example is located in the `mobile` module of the `WearableDataLayer` project. Run the module on the handheld device and start the example app named `GoogleAPIClient Demo`.

1. Make sure that Google Play Services has been added to your app. Verify that the following dependency is in your build.gradle file:

```
compile 'com.google.android.gms:play-services:+'
```

Android Studio should add this dependency automatically.

2. In the manifest, add the following meta-data tag, which is required by Google Play Services. Place it right before the closing application tag (that is, </application>).

Note that the @integer/google_play_services_version resource is automatically added by Google Play Services.

In AndroidManifest.xml:

```
<meta-data
    android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version" />
```

3. Declare the activity.

This is a standard launcher activity declaration with the exception of taskAffinity, which is only needed because this project uses multiple launcher activities that should be treated independently.

In AndroidManifest.xml:

```
<activity
    android:name=".GoogleApiClientActivity"
    android:label="GoogleApiClient Demo"
    android:taskAffinity="com.ocddevelopers.androidwearables.wearabledatalayer.
    ApiTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

4. Declare member variables.

In GoogleApiClientActivity.java:

```
public class GoogleApiClientActivity extends ActionBarActivity {
    private static final String STATE_RESOLVING_ERROR = "resolving_error";
    private static final int REQUEST_RESOLVE_ERROR = 1001;
    private static final String ERROR_FRAGMENT_TAG = "errordialog";
    private boolean mResolvingError = false;
    private GoogleApiClient mGoogleApiClient;
    ...
}
```

5. Implement ConnectionCallbacks, which notifies the app when the connection to GoogleApiClient has been established.

In `GoogleApiClientActivity.java`:

```
private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks =
    new GoogleApiClient.ConnectionCallbacks() {
    @Override
    public void onConnected(Bundle bundle) {
        // you can now use the wearable data layer API
        Toast.makeText(getApplicationContext(), "Connected to GoogleApiClient",
            Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onConnectionSuspended(int cause) {
        // disable any buttons or interface elements that require the wearable API
    }
};
```

The `onConnected` method is a good place to enable any UI element that relies on the connection to `GoogleApiClient`. In turn, the `onConnectionSuspended` method is a good place to disable these UI elements. The connection can be suspended for a variety of reasons which are outside of our control. If the connection is suspended, the app automatically attempts to reconnect to `GoogleApiClient`, after which `onConnected` is called again.

6. Create an `OnConnectionFailedListener` to handle any errors that occur while connecting to Google Play Services.

Many errors can be fixed by user intervention, in which case `connectionResult.hasResolution()` returns true. In this case, start a dialog to prompt the user to solve the problem. Google Play Services takes care of displaying the appropriate dialog when the app calls the `startResolutionForResult` method. For instance, the user can be prompted to install or update Google Play Services if needed. If an error has no resolution, then the app should display an error dialog. Note that the `mResolvingError` flag prevents the code from trying to display an error dialog multiple times, since `onConnectionFailed` may be called multiple times.

In `GoogleApiClientActivity.java`:

```
private GoogleApiClient.OnConnectionFailedListener mConnectionFailedListener =
    new GoogleApiClient.OnConnectionFailedListener() {
    @Override
    public void onConnectionFailed(ConnectionResult connectionResult) {
        if (mResolvingError) {
            // Already attempting to resolve an error.
            return;
        } else if (connectionResult.hasResolution()) {
            try {
                mResolvingError = true;
                connectionResult.startResolutionForResult
                    (GoogleApiClientActivity.this,
                     REQUEST_RESOLVE_ERROR);
            } catch (IntentSender.SendIntentException e) {
```

```

        // There was an error with the resolution intent. Try again.
        mGoogleApiClient.connect();
    }
} else {
    // Show dialog using GooglePlayServicesUtil.getErrorDialog()
    showErrorDialog(connectionResult.getErrorCode());
    mResolvingError = true;
}
}
};

private void showErrorDialog(int errorCode) {
    Dialog errorDialog = GooglePlayServicesUtil.getErrorDialog(errorCode,
        this, REQUEST_RESOLVE_ERROR);

    // Create a fragment for the error dialog
    AlertDialogFragment dialogFragment = AlertDialogFragment.
newInstance(errorDialog,
        new DialogInterface.OnCancelListener() {
            @Override
            public void onCancel(DialogInterface dialog) {
                mResolvingError = false;
            }
        });

    dialogFragment.show(getFragmentManager(), ERROR_FRAGMENT_TAG);
}

```

7. If the user was prompted to resolve an error as described in the previous step, the dialog returns its status to the `onActivityResult` method. If the `resultCode` is `RESULT_OK`, then the error was solved and the app can attempt to connect to `GoogleApiClient` once again.

In `GoogleApiClientActivity.java`:

```

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_RESOLVE_ERROR) {
        mResolvingError = false;
        if (resultCode == RESULT_OK) {
            // Make sure the app is not already connected or attempting to connect
            if (!mGoogleApiClient.isConnecting() &&
                !mGoogleApiClient.isConnected()) {
                mGoogleApiClient.connect();
            }
        }
    }
}
}

```


8. Make the `mResolvingError` flag persistent.

In `GoogleApiClientActivity.java`:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putBoolean(STATE_RESOLVING_ERROR, mResolvingError);
}
```

9. In `onCreate`, initialize the instance of `GoogleApiClient` and restore the value of `mResolvingError` if needed.

In `GoogleApiClientActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addApi(Wearable.API)
        .addConnectionCallbacks(mConnectionCallbacks)
        .addOnConnectionFailedListener(mConnectionFailedListener)
        .build();

    mResolvingError = savedInstanceState != null
    && savedInstanceState.getBoolean(STATE_RESOLVING_ERROR, false);
}
```

10. If the user is not currently trying to resolve an error, connect to and disconnect from `GoogleApiClient` in `onStart` and `onStop`, respectively, to ensure that the connection is only established while the activity is in the foreground.

In `GoogleApiClientActivity.java`:

```
@Override
protected void onStart() {
    super.onStart();
    if(!mResolvingError) {
        mGoogleApiClient.connect();
    }
}

@Override
protected void onStop() {
    if(!mResolvingError) {
        mGoogleApiClient.disconnect();
    }
    super.onStop();
}
```

At this point, you can run to app and verify that it connects with `GoogleApiClient`. Although connecting to `GoogleApiClient` asynchronously requires a lot of code, most of the error handling code is boilerplate, and the code that establishes the connection is surprisingly straightforward in isolation.

A Minimal Asynchronous Connection

The majority of the code for the previous example was tedious error handling. While this tedious code is important for production apps, I recommend using a minimal version of the code to keep things simple while learning. Throughout the rest of the book, we'll use a simplified version of this code.

Note The source code for this example is located in the `MinimalGoogleApiClientActivity.java` class that is contained in the `mobile` module of the `WearableDataLayer` project. This activity does not have a layout and is not declared in the manifest, but instead is included in the project only for convenience.

1. Declare and initialize `GoogleApiClient`.

In `MinimalGoogleApiClientActivity.java`:

```
public class MinimalGoogleApiClientActivity extends Activity {
    private GoogleApiClient mGoogleApiClient;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mGoogleApiClient = new GoogleApiClient.Builder(this)
            .addApi(Wearable.API)
            .addConnectionCallbacks(mConnectionCallbacks)
            .build();
    }
    ...
}
```

2. Connect to `GoogleApiClient` asynchronously from `onStart` and disconnect from `onStop`.

In `MinimalGoogleApiClientActivity.java`:

```
@Override
protected void onStart() {
    super.onStart();
    mGoogleApiClient.connect();
}
```

```
@Override
protected void onStop() {
    mGoogleApiClient.disconnect();
    super.onStop();
}
```

3. Begin using `GoogleApiClient` once the `onConnected` method of the `ConnectionCallbacks` is called.

In `MinimalGoogleApiClientActivity.java`:

```
private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks =
    new GoogleApiClient.ConnectionCallbacks() {
    @Override
    public void onConnected(Bundle bundle) {
        // begin using GoogleApiClient as of here
    }

    @Override
    public void onConnectionSuspended(int cause) {
    }
};
```

Note how straightforward connecting to `GoogleApiClient` is once we get rid of all the error handling code. While activities should typically connect to `GoogleApiClient` asynchronously, `IntentServices` usually establish a connection synchronously.

Establishing a Synchronous Connection

Connecting to Google Play Services synchronously is a lot simpler but can only be used in particular situations. For instance, if you are connecting to Google Play Services from an `IntentService`, you should do so synchronously since the code is not using the UI thread.

1. Create `GoogleApiClient` and add the `Wearable` API. There is no need to set `ConnectionCallbacks` since the connection is established synchronously.

```
GoogleApiClient googleApiClient = new GoogleApiClient.Builder(this)
    .addApi(Wearable.API)
    .build();
```

2. Call `GoogleApiClient`'s `blockingConnect` method and pass it the maximum amount of time you're willing to wait for the connection. Here, we use 10 seconds.

```
ConnectionResult connectionResult = googleApiClient.blockingConnect(10, TimeUnit.
SECONDS);
```

3. Check to see if the connection was successfully established.

```

if(!connectionResult.isSuccess()) {
    Log.e("DK", "Failed to connect to GoogleApiClient.");
    return;
}

```

Now that we know how to connect to `GoogleApiClient`, we'll learn about the Node API.

PendingResults

The wearable data layer API contains many methods that take a non-negligible amount of time to execute. Operations such as fetching a particular piece of data are often slow and should not be executed on the UI thread. All of these methods return a `PendingResult<T>`, which is a generic class that represents a result that has not yet been obtained.

The generic type of a `PendingResult` depends on the particular method that was invoked. The generic type contains methods that retrieve that data of interest. For instance, the Node API's `getLocalNode` method, which we'll learn about shortly, returns an instance of `PendingResult<NodeApi.GetLocalNodeResult>`. The `GetLocalNodeResult` class contains a method called `getNode`, which retrieves the meta-data for the local node.

A `PendingResult` can either wait for the desired data synchronously or asynchronously. Consider a variable called `mResult` of type `PendingResult<T>`:

- Calling `mResult.await()` returns an instance of `T` synchronously. That is, this method blocks until the instance of `T` is available. As a result, this method should never be called on the UI thread.
- Calling `mResult.setResultCallback(...)` specifies a callback that gets notified once the instance of `T` is available. This method is asynchronous and can safely be called from the UI thread. The `setResultCallback` does not return anything since the value of `T` is not available until it's passed into the result callback.

For example, say `mResult` is of type `PendingResult<GetLocalNodeResult>`. The `GetLocalNodeResult` class contains a getter that provides access to the local node (as we'll see in the next section). To retrieve the local node synchronously, call

```

GetLocalNodeResult localNodeResult = mResult.await();
Node localNode = localNodeResult.getNode();
...

```

To retrieve the local node asynchronously, call

```

mResult.setResultCallback(new ResultCallback<NodeApi.GetLocalNodeResult>() {
    @Override
    public void onResult(NodeApi.GetLocalNodeResult localNodeResult) {
        Node localNode = localNodeResult.getNode();
        ...
    }
});

```

The `PendingResult` pattern is used by the Node API, Message API, and Data API. Though the pattern may seem a bit confusing to begin with, being able to choose whether the data is retrieved synchronously or asynchronously is a big advantage.

Example #2: Using the Node API

A node is a device that can communicate using the wearable data layer API. The local node refers to the device on which an app is running on, and the remote node refers to the device that an app communicates with. From the perspective of the handheld app, the handheld device is the local node and the wearable is the remote node. Similarly, from the perspective of the wearable, the handheld device is the remote node and the wearable is the local node.

The most important use of the Node API is monitoring the status of a connection with a remote node. When an app detects that the remote node is disconnected, it can disable user interface elements that depend on the connection. A secondary use of the Node API is to obtain the node IDs for local and remote nodes. Certain methods of the wearable data layer API need these node IDs.

This example app demonstrates how to 1) monitor the connection status of the remote node, 2) obtain the local node's ID, and 3) obtain the remote node's ID. Every node also has a `displayName` property, which this app displays in addition to the node ID. The app runs on a handheld device, but the Node API could be used on an Android Wear device without any changes.

Note The source code for this example is located in the `mobile` module of the `WearableDataLayer` project. Run the module on the handheld device and start the example app named **Node API Demo**.

1. Create a layout.

The first `TextView` of this layout indicates whether a device is connected to a remote node. The rest of the layout displays the node ID and display name for both the local node and the remote node.

In `res > layout > activity_nodeapi.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:padding="16dip"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/connection_status"
        android:text="status unavailable"
```

```
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

<TextView
    android:text="LOCAL NODE DISPLAY NAME"
    android:textAppearance="?android:attr/textAppearanceSmall"
    android:fontFamily="sans-serif-light"
    android:layout_marginTop="16dip"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/local_displayname"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:text="LOCAL NODE ID"
    android:textAppearance="?android:attr/textAppearanceSmall"
    android:fontFamily="sans-serif-light"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/local_nodeid"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:text="REMOTE NODE DISPLAY NAME"
    android:textAppearance="?android:attr/textAppearanceSmall"
    android:fontFamily="sans-serif-light"
    android:layout_marginTop="16dip"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/remote_displayname"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:text="REMOTE NODE ID"
    android:textAppearance="?android:attr/textAppearanceSmall"
    android:fontFamily="sans-serif-light"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/remote_nodeid"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

</LinearLayout>
```

2. Declare the activity.

This is a standard launcher activity declaration with the exception of `taskAffinity`, which is only needed because this project uses multiple launcher activities that should be treated independently.

In `AndroidManifest.xml`:

```
<activity
    android:name=".NodeApiActivity"
    android:label="Node API Demo"
    android:taskAffinity="com.ocddevelopers.androidwearables.wearabledatalayer.
    NodeTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3. Declare and initialize member variables.

In `NodeApiActivity.java`:

```
public class NodeApiActivity extends ActionBarActivity {
    private GoogleApiClient mGoogleApiClient;
    private TextView mConnectionStatus;
    private TextView mLocalDisplayName, mLocalNodeId;
    private TextView mRemoteDisplayName, mRemoteNodeId;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_nodeapi);

        mConnectionStatus = (TextView) findViewById(R.id.connection_status);
        mLocalDisplayName = (TextView) findViewById(R.id.local_displayname);
        mLocalNodeId = (TextView) findViewById(R.id.local_nodeid);
        mRemoteDisplayName = (TextView) findViewById(R.id.remote_displayname);
        mRemoteNodeId = (TextView) findViewById(R.id.remote_nodeid);

        mGoogleApiClient = new GoogleApiClient.Builder(this)
            .addApi(Wearable.API)
            .addConnectionCallbacks(mConnectionCallbacks)
            .build();
    }
    ...
}
```

4. Implement ConnectionCallbacks.

The Node API cannot be utilized until the app has successfully connected to `GoogleApiClient`. The `onConnected` method indicates that the connection was established, and here we use the Node API to add a `NodeListener`, which receives changes in connection status, and to obtain the meta-data of the local and remote nodes.

In `NodeApiActivity.java`:

```
private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks =
    new GoogleApiClient.ConnectionCallbacks() {
        @Override
        public void onConnected(Bundle bundle) {
            Wearable.NodeApi.addListener(mGoogleApiClient, mNodeListener);
            fetchLocalNode();
            fetchRemoteNode();
        }

        @Override
        public void onConnectionSuspended(int cause) {
        }
    };
```

5. Implement NodeListener.

This listener detects changes in the connection status of the remote node. The Node API was designed to handle multiple remote nodes, but the code in this step assumes that there cannot be more than one remote node for simplicity. Therefore, receiving a call to `onPeerConnected` implies that the remote node (that is, the watch or the handheld) just established a connection. The `updateConnectionStatus` method updates the text that indicates whether the remote node is connected.

Note that `NodeListener` only detects changes in connection status and does not indicate whether the remote node is connected to begin with. We will check the initial status of the connection in step 8.

In `NodeApiActivity.java`:

```
private NodeApi.NodeListener mNodeListener = new NodeApi.NodeListener() {
    @Override
    public void onPeerConnected(Node node) {
        // does not get called on UI thread
        updateConnectionStatus(true);
    }

    @Override
    public void onPeerDisconnected(Node node) {
        // does not get called on UI thread
        updateConnectionStatus(false);
    }
};
```


6. Implement the `updateConnectionStatus` method, which updates the UI to indicate whether the device is connected to a remote node.

This method is called from `NodeListener` from a thread that isn't the UI thread. Thus, we must use `runOnUiThread` to ensure that the `TextViews` are modified from the UI thread.

In `NodeApiActivity.java`:

```
private void updateConnectionStatus(final boolean connected) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            if(connected) {
                mConnectionStatus.setText("connected to remote device");
            } else {
                mConnectionStatus.setText("not connected to remote device");
                mRemoteNodeId.setText("");
                mRemoteDisplayName.setText("");
            }
        }
    });
}
```

7. Implement `fetchLocalNode`.

This method uses the Node API's `getLocalNode` method to obtain the meta-data for the local node. The `getLocalNode` method returns an instance of `PendingResult<GetLocalNodeResult>`, and we call its `setResultCallback` to obtain the result asynchronously to avoid blocking the UI thread.

In `NodeApiActivity.java`:

```
private void fetchLocalNode() {
    Wearable.NodeApi.getLocalNode(mGoogleApiClient).setResultCallback(
        new ResultCallback<NodeApi.GetLocalNodeResult>() {
            @Override
            public void onResult(NodeApi.GetLocalNodeResult localNodeResult) {
                Node localNode = localNodeResult.getNode();
                mLocalDisplayName.setText(localNode.getDisplayName());
                mLocalNodeId.setText(localNode.getId());
            }
        });
}
```

8. Implement `fetchRemoteNode`.

This method uses the Node API's `getConnectedNodes` method to obtain a list of nodes that are connected to the current device. Though this API was clearly designed to handle multiple connected nodes, we assume that there cannot be more than a single remote node. If the list of connected nodes is empty, the remote node is disconnected. If, on the other hand, the list is not empty, the

remote node is connected and we obtain its meta-data by fetching the first (and only) node from the list. We then update the UI with the meta-data of the remote node, if available, and the current status of the connection.

In `NodeApiActivity.java`:

```
private void fetchRemoteNode() {
    Wearable.NodeApi.getConnectedNodes(mGoogleApiClient).setResultCallback(
        new ResultCallback<NodeApi.GetConnectedNodesResult>() {
            @Override
            public void onResult(NodeApi.GetConnectedNodesResult result) {
                boolean connected;

                if(result.getNodes().size() > 0) {
                    // assumption: there is no more than 1 connected node
                    Node remoteNode = result.getNodes().get(0);
                    mRemoteDisplayName.setText(remoteNode.getDisplayName());
                    mRemoteNodeId.setText(remoteNode.getId());
                    connected = true;
                } else {
                    connected = false;
                }

                updateConnectionStatus(connected);
            }
        });
}
```

9. Connect and disconnect to `GoogleApiClient` in `onStart` and `onStop`, respectively. Additionally, remove the `NodeListener` in `onStop`.

In `NodeApiActivity.java`:

```
@Override
protected void onStart() {
    super.onStart();
    mGoogleApiClient.connect();
}

@Override
protected void onStop() {
    Wearable.NodeApi.removeListener(mGoogleApiClient, mNodeListener);
    mGoogleApiClient.disconnect();
    super.onStop();
}
```

Run this example app on a handheld and experiment with disabling and enabling the device's Bluetooth to detect changes in the status of the connection with the wearable.

Messages and Data

There are two APIs to communicate between handhelds and wearables: the Message API and the Data API. The Message API lets you send one-way messages from one device to the other. Once the messages are sent, there is no confirmation that they were received. If you attempt to send a message when a connection is unavailable, the message will not be delivered nor will it automatically be resent once the system reconnects. As a result, this API is good for executing remote procedure calls on the other device. The Message API lets you, for instance, pause and play music that's playing on a handheld from Android Wear.

The Data API lets you share and synchronize data between devices. When a user starts a timer on a handheld, for instance, the Data API can transmit the due time to the Android Wear device. If the user stops or modifies the timer from the wearable, the Data API automatically updates handheld device with the most recent information. If a connection is unavailable when data is posted to the Data API, it will automatically synchronize with the other device once the connection is reestablished.

Use the Message API when:

- a device should immediately invoke an action on the other device, such as starting an activity.
- you're primarily interested in one-way communication.
- you don't want the system to manage and cache messages.

Use the Data API when:

- a device needs to synchronize data that may be modified on either device.
- you're interested in one-way or two-way data communication. That is, either device may manipulate data, and it will be automatically synced with the other device.
- you want the system to manage and cache data.

We'll learn about the Message and Data APIs in the context of the counter example app.

Example #3: Building a Shared Counter

This app demonstrates 1) how to use the Message API to start an activity on the wearable by pressing a button on the handheld and 2) how to synchronize a number between both devices with the Data API. The app contains a counter that is displayed on both the handheld and the wearable device and is synchronized across both devices (see Figure 6-1).

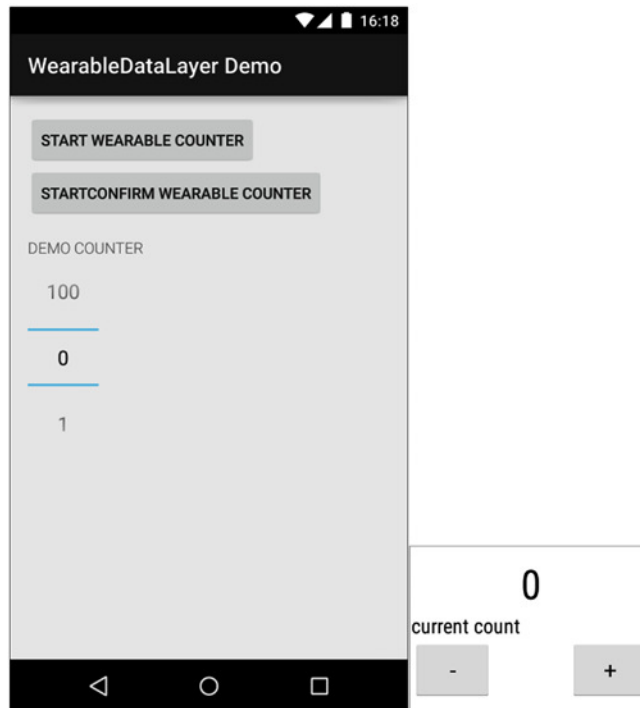


Figure 6-1. The counter demo demonstrates the use of the Message and Data APIs by synchronizing a number across a handheld and a wearable. The handheld app lets users change the count on a number picker (left). The wearable app lets users change the count by tapping on buttons that decrement or increment it (right)

Note The source code for this example is located in the `WearableDataLayer` project, which contains both the mobile and wear modules for the handheld and Android Wear device, respectively. Run each module on its respective device and verify that the devices are paired. Start the example app named `WearableDataLayer Demo` on the handheld device and tap on the first button (which is labeled *Start Wearable Counter*). An activity should pop up on the Android Wear device in response to the button tap on the handheld.

On the handheld app, pressing the *Start Wearable Counter* or the *StartConfirm Wearable Counter* buttons send a message to the wearable, which in turn opens its counter activity. Changing the value of the number picker on the handheld transmits the most recent value to the wearable, where it's displayed in the counter activity.

On the wearable, pressing the buttons send a message to the handheld, which in turn decrements or increments the count and transmits the new value to the wearable. The count data is therefore managed by the handheld.

This section describes the beginning of the implementation of the counter app. The rest of the implementation, which uses the Message and Data APIs, is described in following sections.

Implementing the Handheld Activity

The counter example contains code that resides on both the mobile and the wear modules. This section implements the foundation of MainActivity, which resides on the mobile module and runs on the handheld.

1. Create a layout.

This layout is shown in the left of Figure 6-1.

In the mobile module's res > layout > activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dip">

    <Button
        android:id="@+id/start_wearable"
        android:text="Start Wearable Counter"
        android:onClick="onStartWearableCounterClick"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <Button
        android:id="@+id/start_confirm_wearable"
        android:text="StartConfirm Wearable Counter"
        android:onClick="onStartConfirmWearableCounterClick"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:text="demo counter"
        android:textAppearance="@android:style/TextAppearance.Small"
        android:textAllCaps="true"
        android:fontFamily="sans-serif-light"
        android:layout_marginTop="16dip"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <NumberPicker
        android:id="@+id/activity_main"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

2. Declare the activity.

This is a standard launcher activity declaration with the exception of `taskAffinity`, which is only needed because this project uses multiple launcher activities that should be treated independently.

In the mobile module's `AndroidManifest.xml`:

```
<activity
    android:name=".MainActivity"
    android:label="WearableDataLayer Demo"
    android:taskAffinity="com.ocddevelopers.androidwearables.wearabledatalayer.MainTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3. Declare member variables.

In the mobile module's `MainActivity.java`:

```
public class MainActivity extends ActionBarActivity {
    private static final String START_ACTIVITY_PATH = "/start/CounterActivity";
    private static final String START_CONFIRM_ACTIVITY_PATH = "/start_confirm/CounterActivity";
    private static final String CONFIRMATION_PATH = "/confirm/CounterActivity";
    private static final String COUNTER_PATH = "/counter";
    public static final String COUNTER_INCREMENT_PATH = "/counter/increment";
    public static final String COUNTER_DECREMENT_PATH = "/counter/decrement";
    private static final String KEY_COUNT = "count";
    private int mCount;
    private NumberPicker mNumberPicker;
    private GoogleApiClient mGoogleApiClient;
    ...
}
```

4. Initialize member variables and disable the buttons in `onCreate`.

The buttons should not be enabled until a connection to `GoogleApiClient` has been established.

In the mobile module's `MainActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    mNumberPicker = (NumberPicker) findViewById(R.id.counter);
    mNumberPicker.setMinValue(0);
    mNumberPicker.setMaxValue(100);
    mNumberPicker.setOnValueChangedListener(mValueChangedListener);
}
```

```
mGoogleApiClient = new GoogleApiClient.Builder(this)
    .addApi(Wearable.API)
    .addConnectionCallbacks(mConnectionCallbacks)
    .build();

// disable buttons until GoogleApiClient is connected
setButtonsEnabled(false);

mCount = 0;
}

private void setButtonsEnabled(boolean enabled) {
    findViewById(R.id.start_wearable).setEnabled(enabled);
    findViewById(R.id.start_confirm_wearable).setEnabled(enabled);
}
```

5. Connect to and disconnect from GoogleApiClient in onStart and onStop, respectively.

In the mobile module's MainActivity.java:

```
@Override
protected void onStart() {
    super.onStart();
    mGoogleApiClient.connect();
}

@Override
protected void onStop() {
    mGoogleApiClient.disconnect();
    super.onStop();
}
```

6. Implement ConnectionCallbacks.

Enable the buttons when the onConnected method is called once the connection to GoogleApiClient has been established. If the onConnectionSuspended method is called, disable the buttons.

In the mobile module's MainActivity.java:

```
private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks =
    new GoogleApiClient.ConnectionCallbacks() {
    @Override
    public void onConnected(Bundle bundle) {
        fetchCurrentCount();
        setButtonsEnabled(true);
    }

    @Override
    public void onConnectionSuspended(int cause) {
        setButtonsEnabled(false);
    }
};
```

7. Handle button clicks.

For now, these methods will be empty.

In the mobile module's `MainActivity.java`:

```
public void onStartWearableCounterClick(View view) {
}

public void onStartConfirmWearableCounterClick(View view) {
}
```

At this point, you should be able to run the activity on the handheld, though it won't respond to button presses and it won't communicate with the wearable yet.

Implementing the Wearable Activity

The counter example contains code that resides on both the mobile and the wear modules. This section implements the foundation of `CounterActivity`, which resides on the wear module and runs on the Android Wear device.

1. Create a layout.

This layout is shown on the right of Figure 6-1 and is wrapped by `BoxInsetLayout` to ensure that it displays correctly on both square and round watches.

In the wear module's `res > layout > activity_counter.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <RelativeLayout
        app:layout_box="all"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:id="@+id/count"
            android:textSize="32sp"
            android:layout_centerInParent="true"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <TextView
            android:id="@+id/count_label"
            android:text="current count"
            android:layout_below="@+id/count"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
    </RelativeLayout>
</android.support.wearable.view.BoxInsetLayout>
```



```

<Button
    android:id="@+id/decrement"
    android:text="-"
    android:onClick="onDecrementClick"
    android:layout_below="@id/count_label"
    android:layout_alignParentLeft="true"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<Button
    android:id="@+id/increment"
    android:text="+"
    android:onClick="onIncrementClick"
    android:layout_below="@id/count_label"
    android:layout_alignParentRight="true"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

</RelativeLayout>
</android.support.wearable.view.BoxInsetLayout>

```

2. Declare the activity.

In the final implementation, this activity is started from the handheld app, so there is no need to give it a voice action.

In the wear module's `CounterActivity.java`:

```

<activity
    android:name=".CounterActivity"
    android:label="Counter Activity" />

```

3. Declare constants and member variables.

In the wear module's `CounterActivity.java`:

```

public class CounterActivity extends Activity {
    public static final String COUNTER_INCREMENT_PATH = "/counter/increment";
    public static final String COUNTER_DECREMENT_PATH = "/counter/decrement";
    public static final String COUNTER_PATH = "/counter";
    public static final String KEY_COUNT = "count";
    private TextView mCountText;
    private GoogleApiClient mGoogleApiClient;
    int mCount = -1;
    ...
}

```

4. Initialize member variables.

In the wear module's `CounterActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_counter);

    mCountText = (TextView) findViewById(R.id.count);

    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addApi(Wearable.API)
        .addConnectionCallbacks(mConnectionCallbacks)
        .build();
}
```

5. Connect to and disconnect from `GoogleApiClient` in `onStart` and `onStop`, respectively.

In the wear module's `CounterActivity.java`:

```
@Override
protected void onStart() {
    super.onStart();
    mGoogleApiClient.connect();
}

@Override
protected void onStop() {
    mGoogleApiClient.disconnect();
    super.onStop();
}
```

6. Handle button taps.

These methods are empty for now.

In the wear module's `CounterActivity.java`:

```
public void onIncrementClick(View view) {
}

public void onDecrementClick(View view) {
}
```

7. Implement the `NumberPicker`'s `ONValueChangeListener`.

We'll leave this listener empty for now.

In the wear module's `CounterActivity.java`:

```
private NumberPicker.OnValueChangeListener mValueChangeListener = new
NumberPicker.OnValueChangeListener() {
    @Override
    public void onValueChange(NumberPicker picker, int oldVal, int newVal) {
    }
};
```

Now that we've implemented the foundation of the counter app on both handheld and wearable, we can implement additional features using the Message API.

Sending Messages with the Message API

Messages are a one-way communication mechanism that are good for remote procedure calls such as sending a message to the wearable to start an activity. If necessary, you can respond to a message with another message for a request/response paradigm, but typical communication occurs in one direction.

To send a message, find the node ID of the destination and use it as a parameter in the `Wearable.MessageApi.sendMessage` method, which takes four parameters:

- an instance of `GoogleApiClient`
- the ID of the destination node
- the path of the message
- a byte array of data to send with the message

Every message is identified by a path, which is a unique string that starts with a forward slash, such as `/path/to/data`. Paths allow devices that receive messages to identify which message has been sent. Messages can transmit a byte array of data, but many messages do not have any data to transmit and simply pass this parameter null.

To demonstrate the Message API, we'll start `CounterActivity` on Android Wear by tapping on a button on handheld device's `MainActivity`, which sends a message to the wearable.

In the mobile module's `MainActivity.java`:

```
public void onStartWearableCounterClick(View view) {
    Wearable.NodeApi.getConnectedNodes(mGoogleApiClient).setResultCallback(
        new ResultCallback<NodeApi.GetConnectedNodesResult>() {
            @Override
            public void onResult(NodeApi.GetConnectedNodesResult getConnectedNodesResult) {
                for (Node node : getConnectedNodesResult.getNodes()) {
                    Wearable.MessageApi.sendMessage(mGoogleApiClient, node.getId(),
                        START_ACTIVITY_PATH, null);
                }
            }
        });
}
```

In this example, we send a message to every connected node, which in our example means just the wearable. Currently, there cannot be more than one connected node.

Now that we're sending a message on the handheld, let's see how to receive it on the wearable.

Receiving Messages with the Message API

We are sending the wearable a message to start an activity. Messages can be received by activities or services that are currently running, but we should be able to start the activity even when there is nothing running to begin with. We can do that with a `WearableListenerService`.

A `WearableListenerService` is a service that can process messages and data sent to your app. `WearableListenerServices` are managed by Android Wear to ensure that they operate efficiently and do not consume excessive resources. We'll create a `WearableListenerService` on the wearable that starts an activity in response to the handheld's message.

1. Declare constants.

These constants are also used in subsequent sections.

In the wear module's `DataLayerListenerService.java`:

```
public class DataLayerListenerService extends WearableListenerService {
    private static final String START_ACTIVITY_PATH = "/start/CounterActivity";
    private static final String START_CONFIRM_ACTIVITY_PATH = "/start_confirm/CounterActivity";
    private static final String CONFIRMATION_PATH = "/confirm/CounterActivity";
    private static final String IMAGE_ASSET_KEY = "preview";
    public static final String TAKE_IMAGE_PATH = "/image";
    ...
}
```

2. Register the `WearableListenerService` in the manifest.

The `BIND_LISTENER` intent-filter tells Android Wear that this service is a `WearableListenerService` and that it should receive messages and data.

In the wear module's `AndroidManifest.xml`:

```
<service android:name=".DataLayerListenerService">
    <intent-filter>
        <action android:name="com.google.android.gms.wearable.BIND_LISTENER" />
    </intent-filter>
</service>
```

3. Implement `onMessageReceived` and start `CounterActivity` when the message with path `START_ACTIVITY_PATH` is received.

The activity is started with the flags 'new task' and 'single top' to allow the activity to popup without any user intervention and to ensure that only a single instance of the activity lies at the top of the activity stack.

In the wear module's `DataLayerListenerService.java`:

```
@Override
public void onMessageReceived(MessageEvent messageEvent) {
    super.onMessageReceived(messageEvent);
    String path = messageEvent.getPath();

    if(START_ACTIVITY_PATH.equals(path)) {
        Intent intent = new Intent(this, CounterActivity.class);
        intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        intent.addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);
        startActivity(intent);
    }
}
```

You should now be able to start `CounterActivity` on Android Wear by tapping on the first button of the handheld app.

Request Response Paradigm

Although messages allow for one-way communication, a device can reply to a message with another message to create two-way communication. We'll demonstrate this by starting an activity in the wearable from the handheld and, in addition, sending the handheld a message to verify that the original message was received.

Note The source code for this example is located in the `WearableDataLayer` project, which contains both the mobile and wear modules for the handheld and Android Wear device, respectively. Run each module on its respective device and verify that the devices are paired. Start the example app named `WearableDataLayer Demo` on the handheld device and tap on the second button (which is labeled `StartConfirm Wearable Counter`). An activity should pop up on the Android Wear device in response to the button tap on the handheld, after which the handheld will display a Toast to confirm that a response was received.

1. Send a message to the wearable when users tap on the *StartConfirm Wearable Counter* button.

This step sends a message just like the previous section, with the exception that the message has a different path.

In the mobile module's `MainActivity.java`:

```
public void onStartConfirmWearableCounterClick(View view) {
    Wearable.NodeApi.getConnectedNodes(mGoogleApiClient).setResultCallback(
        new ResultCallback<NodeApi.GetConnectedNodesResult>() {
```

```

@Override
public void onResult(NodeApi.GetConnectedNodesResult
getConnectedNodesResult) {
    for (Node node : getConnectedNodesResult.getNodes()) {
        Wearable.MessageApi.sendMessage(mGoogleApiClient, node.getId(),
            START_CONFIRM_ACTIVITY_PATH, null);
    }
}
});
}

```

2. When the wearable receives the message, start the activity and send another message back to the handheld to confirm that the first message was received.

When the wearable receive the message with path `START_CONFIRM_ACTIVITY_PATH`, we start the activity just as in the previous section. The difference here is that we now send a message back to the handheld with the `confirmStarted` method. Add the code in bold to `DataLayerListenerService`'s `onMessageReceived` method.

In the wear module's `DataLayerListenerService.java`:

```

@Override
public void onMessageReceived(MessageEvent messageEvent) {
    super.onMessageReceived(messageEvent);
    String path = messageEvent.getPath();

    if(START_ACTIVITY_PATH.equals(path)) {
        Intent intent = new Intent(this, CounterActivity.class);
        intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        intent.addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);
        startActivity(intent);
    } else if(START_CONFIRM_ACTIVITY_PATH.equals(path)) {
        Intent intent = new Intent(this, CounterActivity.class);
        intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        startActivity(intent);
        confirmStarted(messageEvent.getSourceNodeId());
    }
}
}

```

3. Implement `confirmStarted`.

This method connects to `GoogleApiClient` synchronously and then sends the response message. The `onMessageReceived` method is not called from the UI thread, so establishing a connection synchronously is not a problem.

In the wear module's `DataLayerListenerService.java`:

```

private void confirmStarted(String sourceNodeId) {
    GoogleApiClient googleApiClient = new GoogleApiClient.Builder(this)
        .addApi(Wearable.API)
        .build();
}

```

```

        ConnectionResult connectionResult = googleApiClient.blockingConnect(10,
            TimeUnit.SECONDS);
        if(connectionResult.isSuccess()) {
            Wearable.MessageApi.sendMessage(googleApiClient, sourceNodeId,
                CONFIRMATION_PATH, null);
        }
    }
}

```

4. In the mobile module, register a `MessageListener` with the `Message API` to receive messages.

Add the code in bold to the `onConnected` method.

In the mobile module's `MainActivity.java`:

```

private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks =
    new GoogleApiClient.ConnectionCallbacks() {
        @Override
        public void onConnected(Bundle bundle) {
            Wearable.MessageApi.addListener(mGoogleApiClient, mMessageListener);
            setButtonsEnabled(true);
        }

        @Override
        public void onConnectionSuspended(int cause) {
            setButtonsEnabled(false);
        }
    };

```

5. Implement `MessageListener` and show a `Toast` when the confirmation message is received.

In the mobile module's `MainActivity.java`:

```

private MessageApi.MessageListener mMessageListener = new MessageApi.
MessageListener() {
    @Override
    public void onMessageReceived(MessageEvent messageEvent) {
        if (CONFIRMATION_PATH.equals(messageEvent.getPath())) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Toast.makeText(MainActivity.this, "Wearable CounterActivity
                    started",
                        Toast.LENGTH_SHORT).show();
                }
            });
        }
    }
}

```

6. Since we are adding the `MessageListener` to the `Message` API, we should also remove it when we no longer need it.

In the mobile module's `MainActivity.java`:

```
@Override
protected void onStop() {
    Wearable.MessageApi.removeListener(mGoogleApiClient, mMessageListener);
    mGoogleApiClient.disconnect();
    super.onStop();
}
```

We just covered the basics of the `Message` API. It's time to move on to the `Data` API.

The Data API

The `Data` API defines the interface that the system uses to synchronize data between handhelds and wearables and is based on `DataItems`. `DataItems` are shared among devices and contain small amounts of data. A `DataItem` has two parts:

1. **Path:** Just like with the `Message` API, a path is unique string that starts with a forward slash such as `/path/to/data`.
2. **Payload:** a byte array limited to 100KB (holds the actual data)

Transmitting data as a byte array requires object serialization and deserialization.

Note Serialization (also known as marshaling) is the procedure of translating an object or data structure into a byte array (or another form suitable for storing or transmitting, such as text). Deserialization (also known as unmarshaling) is the reverse procedure of translating a byte array into a data structure. Objects and data structures cannot be transmitted over a network connection (such as the data layer API) unless they're serialized.

Alas, serializing objects is not very fun. Fortunately, the SDK can serialize the data for us with the `DataMap` class, which serializes data given as an `Android Bundle`. A common way of doing so is with the `PutDataMapRequest` class. The following code demonstrates how to put data into a `PutDataMapRequest` and is not part of the counter example:

3. Create a `PutDataMapRequest` with the path of the desired data item.

```
PutDataMapRequest dataMap = PutDataMapRequest.create("/message");
```


4. Obtain a `DataMap` by calling `PutDataMapRequest.getDataMap()` and add values just as you would with a `Bundle`.

```
dataMap.getDataMap().putInt(KEY_NUMBER, 42);
```

5. Obtain a `PutDataRequest` instance.

```
PutDataRequest request = dataMap.asPutDataRequest();
```

6. Insert the `PutDataRequest` into the data layer.

```
PendingResult<DataApi.DataItemResult> pendingResult = Wearable.DataApi.  
putDataItem(mGoogleApiClient, request);
```

Note If the handset and wearable devices are disconnected, the data layer caches data and syncs when the connection is re-established.

We'll demonstrate the Data API by synchronizing a counter between the handheld and wearable. You can increment or decrement the counter on either device and they will remain in sync. The best way to synchronize data is to pick a single device to manage it. In other words, you should only update a data item from a single device.

In this case, the handheld app manages the data item that contains the count. When users decrement or increment the count from the wearable app, it sends messages to the handheld app, which in turn updates the data item. The wearable then receives the data item and updates the count.

Note The source code for this example is located in the `WearableDataLayer` project, which contains both the mobile and wear modules for the handheld and Android Wear device, respectively. Run each module on its respective device and verify that the devices are paired. Start the example app named `WearableDataLayer Demo` on the handheld device and tap on the first button (which is labeled *Start Wearable Counter*). An activity should pop up on the Android Wear device. You can then increment or decrement the shared counter from either device.

7. On the wearable, add a `DataListener` to the Data API to receive data items.

Add the code in bold to the `onConnected` method.

In the wear module's `CounterActivity.java`:

```
private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks =
    new GoogleApiClient.ConnectionCallbacks() {
        @Override
        public void onConnected(Bundle bundle) {
            Wearable.DataApi.addListener(mGoogleApiClient, mDataListener);
            loadRemoteNodeId();
        }

        @Override
        public void onConnectionSuspended(int cause) {
        }
    };
```

8. Find the initial value of the count.

The `DataListener` only listens for changes in data items. In other words, it does not receive the initial value of data items that already exist when the activity starts. To obtain the initial value, we'll use the Data API's `getDataItems` method.

In the wear module's `CounterActivity.java`:

```
private void loadRemoteNodeId() {
    Wearable.DataApi.getDataItems(mGoogleApiClient).setResultCallback(new
    ResultCallback<DataItemBuffer>() {
        @Override
        public void onResult(DataItemBuffer dataItems) {
            for (int i = 0; i < dataItems.getCount(); ++i) {
                DataItem dataItem = dataItems.get(i);
                if (COUNTER_PATH.equals(dataItem.getUri().getPath())) {
                    updateCountFromDataItem(dataItem);
                }
            }
        }
    });
}
```

9. Extract the count from the data item received in the previous step and use it to update the `TextView`.

The `updateCountFromDataItem` method obtains a `DataMap` from the data item and uses it to fetch the value of the counter. This method does not always get called from the UI thread, so it uses `runOnUiThread` to ensure it modifies the `TextView` only from the UI thread.

In the wear module's `CounterActivity.java`:

```
private void updateCountFromDataItem(DataItem dataItem) {
    DataMap dataMap = DataMapItem.fromDataItem(dataItem).getDataMap();
    mCount = dataMap.getInt(KEY_COUNT);
    runOnUiThread(new Runnable() {
```

```

        @Override
        public void run() {
            mCountText.setText(Integer.toString(mCount));
        }
    });
}

```

10. Implement DataListener.

The previous steps update the counter as soon as the activity is started. `DataListener` ensures that the counter remains in sync when its value changes after the activity is started. Note that the `freezeIterable` call takes the `DataEventBuffer` and returns an immutable list of `DataEvents`.

In the wear module's `CounterActivity.java`:

```

private DataApi.DataListener mDataListener = new DataApi.DataListener() {
    @Override
    public void onDataChanged(DataEventBuffer dataEvents) {
        final List<DataEvent> events = FreezableUtils
            .freezeIterable(dataEvents);
        for(DataEvent dataEvent : events) {
            DataItem dataItem = dataEvent.getDataItem();
            if("/counter".equals(dataItem.getUri().getPath())) {
                updateCountFromDataItem(dataItem);
            }
        }
    }
};

```

11. Remove the DataListener once it's no longer in use.

Add the code in bold to the `onStop` method.

In the wear module's `CounterActivity.java`:

```

@Override
protected void onStop() {
    Wearable.DataApi.removeListener(mGoogleApiClient, mDataListener);
    mGoogleApiClient.disconnect();
    super.onStop();
}

```

12. In the handheld module, send a data item any time users change the count on the NumberPicker.

Add the code in bold to `onValueChange`.

In the handheld module's `MainActivity.java`:

```

private NumberPicker.OnValueChangeListener mValueChangeListener = new
NumberPicker.OnValueChangeListener() {

```

```

@Override
public void onValueChange(NumberPicker picker, int oldVal, int newVal) {
    // push the new value of the counter to the Data API
    mCount = newVal;
    updateCountData();
}
};

```

13. In the handheld module, implement `updateCountData`.

This method updates the data item.

In the handheld module's `MainActivity.java`:

```

private void updateCountData() {

    PutDataMapRequest updateCounterDataMapRequest = PutDataMapRequest.
        create(COUNTER_PATH);
    updateCounterDataMapRequest.getDataMap().putInt(KEY_COUNT, mCount);
    PutDataRequest putDataRequest = updateCounterDataMapRequest.
        asPutDataRequest();

    PendingResult<DataApi.DataItemResult> pendingResult =
        Wearable.DataApi.putDataItem(mGoogleApiClient, putDataRequest);
    pendingResult.setResultCallback(new ResultCallback<DataApi.DataItemResult>() {
        @Override
        public void onResult(DataApi.DataItemResult dataItemResult) {
        }
    });
}

```

If you run the example now, you should see the wearable display the count value as it's updated from the handheld. The next section implements the decrement and increment buttons on the wearable.

Implementing the Decrement and Increment Buttons

These buttons on the wearable should send messages to the handheld device, which should in turn update the counter and its data item.

1. When a button is tapped, send a message to the handheld device.

In the wear module's `CounterActivity.java`:

```

public void onIncrementClick(View view) {
    sendMessage(COUNTER_INCREMENT_PATH);
}

public void onDecrementClick(View view) {
    sendMessage(COUNTER_DECREMENT_PATH);
}

```

2. Implement sendMessage.

This method sends a message to the wearable indicating that the count should be decremented or incremented.

In the wear module's CounterActivity.java:

```
private void updateCount(final String action) {
    Wearable.NodeApi.getConnectedNodes(mGoogleApiClient).setResultCallback(
        new ResultCallback<NodeApi.GetConnectedNodesResult>() {
            @Override
            public void onResult(NodeApi.GetConnectedNodesResult
                getConnectedNodesResult) {
                for (Node node : getConnectedNodesResult.getNodes()) {
                    Wearable.MessageApi.sendMessage(mGoogleApiClient, node.getId(),
                        action, null);
                }
            }
        });
}
```

3. In the handheld, update the number picker and the data item when the app receives a message indicating that the decrement or increment button was pressed on the wearable.

Add the code in bold to the MessageListener.

In the mobile module's MainActivity.java:

```
private MessageApi.MessageListener mMessageListener = new MessageApi.
MessageListener() {
    @Override
    public void onMessageReceived(MessageEvent messageEvent) {
        if (CONFIRMATION_PATH.equals(messageEvent.getPath())) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Toast.makeText(MainActivity.this, "Wearable CounterActivity
                        started",
                        Toast.LENGTH_SHORT).show();
                }
            });
        } else if (COUNTER_INCREMENT_PATH.equals(messageEvent.getPath())) {
            ++mCount;
            updateNumberPicker();
            updateCountData();
        } else if (COUNTER_DECREMENT_PATH.equals(messageEvent.getPath())) {
            --mCount;
            updateNumberPicker();
            updateCountData();
        }
    }
};
```

4. Implement `updateNumberPicker`.

Use `runOnUiThread` to verify that the number picker is updated from the UI thread.

In the mobile module's `MainActivity.java`:

```
private void updateNumberPicker() {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            mNumberPicker.setValue(mCount);
        }
    });
}
```

The example is now complete, and you should be able to modify the counter from both devices. If the devices are not paired, only the handheld device is able to modify the counter. Once the devices are once again paired, the wearable automatically receives the latest data item.

Example #4: Sending and Receiving Assets

Even though the Data API cannot send a payload greater than 100 kb, there are situations in which sharing assets between devices is desirable. For instance, a social media app could download an image on the handheld device and transmit it to the Android Wear device, where it could be displayed as a `BigPictureStyle` notification.

Sending Assets with the Data API

Assets such as images can, fortunately, be transmitted with the Data API.

1. Create a layout.

In `res > layout > activity_sendimage`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/send_image"
        android:text="Send Image"
        android:onClick="onSendImageClick"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

2. Declare the activity.

This is a standard launcher activity declaration with the exception of `taskAffinity`, which is only needed because this project uses multiple launcher activities that should be treated independently.

In the mobile module's `AndroidManifest.xml`:

```
<activity
    android:name=".SendImageActivity"
    android:label="Send Image Demo"
    android:taskAffinity="com.ocddevelopers.androidwearables.wearabledatalayer.
    ImageTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3. Declare variables.

In the mobile module's `SendImageActivity.java`:

```
public class SendImageActivity extends ActionBarActivity {
    private static final int REQUEST_TAKE_PHOTO = 1;
    private static final String TAKE_IMAGE_PATH = "/image";
    private static final String IMAGE_ASSET_KEY = "preview";
    private String mCurrentPhotoPath;
    private GoogleApiClient mGoogleApiClient;
    ...
}
```

4. Initialize, start, and stop `GoogleApiClient`.

We've implemented this step in all the previous examples, so we won't explain it in detail again.

In the mobile module's `SendImageActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_sendimage);

    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addApi(Wearable.API)
        .addConnectionCallbacks(mConnectionCallbacks)
        .build();

    // disable buttons until GoogleApiClient is connected
    setButtonEnabled(false);
}

private void setButtonEnabled(boolean enabled) {
    findViewById(R.id.send_image).setEnabled(enabled);
}
```

```

@Override
protected void onStart() {
    super.onStart();
    mGoogleApiClient.connect();
}

@Override
protected void onStop() {
    mGoogleApiClient.disconnect();
    super.onStop();
}

private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks =
    new GoogleApiClient.ConnectionCallbacks() {
        @Override
        public void onConnected(Bundle bundle) {
            setButtonEnabled(true);
        }

        @Override
        public void onConnectionSuspended(int cause) {
            setButtonEnabled(false);
        }
    };

```

5. When the user taps on the “Send Image,” start an intent that captures an image using the native camera app.

The intent requires the `MediaStore.OUTPUT_URI` extra, which specifies the URI that the image should be saved in.

In the mobile module’s `SendImageActivity.java`:

```

public void onSendImageClick(View view) {
    Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);

    if (takePictureIntent.resolveActivity(getPackageManager()) != null) {
        // Create the File where the photo should go
        File photoFile = null;
        try {
            photoFile = createImageFile();
        } catch (IOException ex) {
            // Error occurred while creating the File
            ex.printStackTrace();
        }
        // Continue only if the File was successfully created
        if (photoFile != null) {
            takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
                Uri.fromFile(photoFile));
            startActivityForResult(takePictureIntent, REQUEST_TAKE_PHOTO);
        }
    }
}

```


6. Implement `createImageFile`, which specifies the location in which the native camera app should save the image.

In the mobile module's `SendImageActivity.java`:

```
private File createImageFile() throws IOException {
    // Create an image file name
    String imageFileName = "wear_image";
    File storageDir = Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_PICTURES);
    File image = File.createTempFile(
        imageFileName, /* prefix */
        ".jpg",        /* suffix */
        storageDir     /* directory */
    );

    // Save a file: path for use with ACTION_VIEW intents
    mCurrentPhotoPath = image.getAbsolutePath();
    return image;
}
```

7. When the native camera app captures a picture, use it to create an asset and transmit it to the Data API.

In the mobile module's `SendImageActivity.java`:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_TAKE_PHOTO && resultCode == RESULT_OK) {
        Bitmap imageBitmap = getPicture();
        Asset asset = createAssetFromBitmap(imageBitmap);
        PutDataRequest request = PutDataRequest.create(TAKE_IMAGE_PATH);
        request.putAsset(IMAGE_ASSET_KEY, asset);
        Wearable.DataApi.putDataItem(mGoogleApiClient, request);
    }
}
```

8. Implement `getPicture`.

This method extracts a bitmap from the path in which the camera app stored it. The method also resizes the image to a size appropriate for a watch. Transmitting the image without resizing it beforehand would be inefficient and unnecessarily consume power. The image gets resized to the dimensions that give the smallest picture larger than 400x400 that can be formed with a subsampling of the original image.

In the mobile module's `SendImageActivity.java`:

```
private Bitmap getPicture() {
    // the desired dimensions of the View
    int targetW = 400;
    int targetH = 400;
```

```

// Get the dimensions of the bitmap
BitmapFactory.Options bmOptions = new BitmapFactory.Options();
bmOptions.inJustDecodeBounds = true;
BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);
int photoW = bmOptions.outWidth;
int photoH = bmOptions.outHeight;

// Determine how much to scale down the image
int scaleFactor = Math.min(photoW/targetW, photoH/targetH);

// Decode the image file into a Bitmap sized to fill the View
bmOptions.inJustDecodeBounds = false;
bmOptions.inSampleSize = scaleFactor;
bmOptions.inPurgeable = true;

return BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);
}

```

Now that we're transmitting an image from the handheld to the wearable, we'll implement the code that receives the image on the wearable.

Retrieving Assets with the Data API

When the wearable receives the image from the handheld, it should display it in a `BigPictureStyle` notification.

1. When the `WearableListenerService` receives the data item for the asset, use `PutDataRequest`'s `getAsset` method to obtain an instance of the `Asset` class that can be used to fetch the bitmap.

In the wear module's `DataLayerListenerService.java`:

```

@Override
public void onDataChanged(DataEventBuffer dataEvents) {
    super.onDataChanged(dataEvents);
    final List<DataEvent> events = FreezableUtils
        .freezeIterable(dataEvents);

    for(DataEvent dataEvent : events) {
        DataItem dataItem = dataEvent.getDataItem();
        String path = dataItem.getUri().getPath();

        if (TAKE_IMAGE_PATH.equals(path)) {
            PutDataRequest request = PutDataRequest.createFromDataItem(dataItem);
            Asset profileAsset = request.getAsset(IMAGE_ASSET_KEY);

            Bitmap image = loadBitmapFromAsset(profileAsset);
            Notification notification = createNotification(image);
        }
    }
}

```

```

        NotificationManagerCompat notificationManager =
            NotificationManagerCompat.from(this);
        notificationManager.notify(1, notification);
    }
}

```

2. Implement `loadBitmapFromAsset`.

This method takes an asset and returns a bitmap. It first connects to `GoogleApiClient` synchronously and then uses the `DataApi.getFdForAsset` to obtain an `InputStream` for the bitmap.

In the wear module's `DataLayerListenerService.java`:

```

public Bitmap loadBitmapFromAsset(Asset asset) {
    if (asset == null) {
        throw new IllegalArgumentException("Asset must be non-null");
    }

    GoogleApiClient googleApiClient = new GoogleApiClient.Builder(this)
        .addApi(Wearable.API)
        .build();

    ConnectionResult result =
        googleApiClient.blockingConnect(500, TimeUnit.MILLISECONDS);
    if (!result.isSuccess()) {
        return null;
    }

    // convert asset into a file descriptor and block until it's ready
    InputStream assetInputStream = Wearable.DataApi.getFdForAsset(
        googleApiClient, asset).await().getInputStream();
    googleApiClient.disconnect();

    if (assetInputStream == null) {
        return null;
    }
    // decode the stream into a bitmap
    return BitmapFactory.decodeStream(assetInputStream);
}

```

3. Create a `BigPictureStyle` notification that displays the image.

In the wear module's `DataLayerListenerService.java`:

```

private Notification createNotification(Bitmap image) {
    NotificationCompat.BigPictureStyle bigPictureStyle
        = new NotificationCompat.BigPictureStyle()
            .bigPicture(image);
}

```

```
        return new NotificationCompat.Builder(this)
            .setContentTitle("Image Received")
            .setContentText("")
            .setSmallIcon(R.drawable.ic_stat_notify)
            .setStyle(bigPictureStyle)
            .build();
    }
```

You can now run the example and transmit an image from the handheld to the wearable device. Note that there is a bit of a delay between the time in which the image is sent and received.

Summary

We learned how to communicate between a handheld and a wearable using both the Message and Data APIs. We learned how to send and receive messages and data items from both activities and `WearableListenerServices`. The counter example, in particular, demonstrated back-and-forth communication between devices. The wearable data layer allows Android Wear to leverage the increased capabilities and larger form factor of a handheld when appropriate. In the next chapter, we'll learn to implement custom watch faces.

Creating Custom Watch Faces

The primary feature of any watch is its ability to tell time. With traditional watches, the watch face is essentially immutable: the information that it contains and its appearance cannot be changed. As a result, many people own multiple watches that they use in different situations, depending on the functionality and appearance that is appropriate for each situation. In contrast, users can change an Android Wear device's watch face at any time with little effort. An Android Wear watch can display an elegant and minimal watch face one moment and a casual watch face full of features the next. Furthermore, these devices can display information that would be inaccessible in traditional watches, such as the number of steps a user has taken or the current weather.

The first version of Android Wear was based on Android version 4.4 and had no official API for developers to make custom watch faces. The next version of Android Wear was based on Android 5.0 and includes an official API that lets developers create custom watch faces that integrate seamlessly with Android Wear. Users install custom watch faces on their phones, just like any other mobile app. The Android Wear companion app then automatically transmits the watch face to the watch.

Note Some developers reverse-engineered the watch faces included with Android Wear and developed custom watch faces with unofficial APIs for the first version of Android Wear. When Google released the official API, these developers were expected to migrate to the official API.

In this chapter, we'll learn to use the watch face API and we'll build three watch faces that showcase a variety of techniques.

The next sections discuss the nuances of a watch face and its different modes of operation.

The Anatomy of a Watch Face

Android Wear devices are built by different manufacturers that use screens of different shapes, sizes, and types. Watch faces should be flexible enough to work with all of these screens, regardless of whether they are square or round, small or large, or LCD or OLED. This section covers the different types of screens and modes that watch faces should support.

Interactive and Ambient modes

Since watches are small devices that should fit comfortably on a user's wrist, they cannot contain large batteries. For instance, the Moto 360 and the LG G watch have battery capacities of 320 mah and 400 mah, respectively.

Note A milliamp-hour (mah) is a unit that quantifies the capacity of a battery. A capacity of 1 mah indicates that the battery can produce a current of 1 milliamp for an entire hour. For reference, an alkaline AA battery typically has a capacity of about 2500mah and the Nexus 6, iPhone 6, and Galaxy Note 4 have batteries with a capacity of 3220 mah.

Some watches, such as the Moto 360, need to be recharged every day or two. A watch's screen consumes power at a relatively high rate, and minimizing its use significantly increases battery life.

Ambient Mode

Ideally, a watch should always display the time so users can see it as soon as they look at the screen, but constantly keeping the screen on severely hinders battery life. Instead, Android Wear displays a simplified version of a watch face that consumes less power because it uses a reduced set of colors and updates the screen no more than once a minute. This mode of operation is called ambient mode. Watch faces in ambient mode should be limited to black, white, and shades of gray.

Interactive Mode

When users touch the screen of the watch or move their wrists to look at the time, a watch face switches to interactive mode, where it renders in full color and updates the screen once a second or faster. For instance, a watch face in interactive mode can draw a second hand that moves at a steady rate with a fluid animation. Although interactive mode consumes power at a relatively high rate, the watch face automatically changes back to ambient mode after a short period of inactivity (see Figure 7-1).

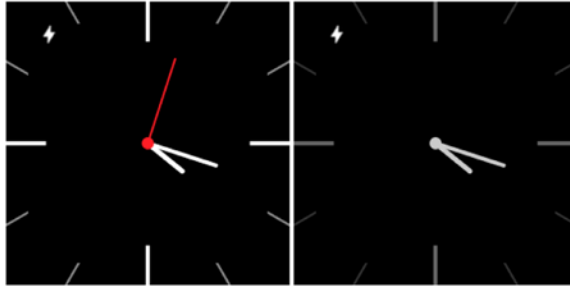


Figure 7-1. A watch face for Android Wear. In interactive mode, the watch face renders in full color and updates the screen at least once a second (left). In ambient mode, the watch face does not display a second hand because it updates once a minute. Additionally, the color palette in ambient mode is limited to black, white, and shades of gray (right)

Why Ambient Mode?

The key point is that a watch face renders in interactive mode when Android Wear detects that a user is looking at the screen. A corollary is that a watch face renders in ambient mode when Android Wear does not detect a user looking at the screen. What is the point, then, of displaying a watch face when a user is not looking at the screen? In reality, ambient mode is not essential. A user can turn off ambient mode altogether in Android Wear’s settings, in which case the watch’s screen will turn off unless it’s in interactive mode. Disabling ambient mode further reduces battery consumption and can be especially useful in watches with small batteries such as the Moto 360. However, keeping ambient mode enabled has its benefits:

- Android Wear does not always correctly identify when a user is looking at the screen. Ambient mode ensures that a user can see the time even when interactive mode is enabled with a delay or is not enabled at all.
- Users are still able to see the time if interactive mode times out when they are still looking at the watch.
- Other people are able to see the watch face even when users themselves are not looking at their watches. Some people give this benefit significant value while others find it irrelevant—it’s a personal preference.

Ensuring that a watch face properly implements ambient mode is the developer’s responsibility. While a poorly implemented watch face can break the rules and use colors or animations in ambient mode, doing so will unnecessarily deplete a watch’s battery. Certain types of screens impose additional constraints on ambient mode as discussed below.

Low-bit Ambient Mode and Burn-in Protection

Two types of screens that are commonly used in Android Wear devices are LCD and OLED. Each type of screen has its inherent benefits and detriments, so neither type is superior to the other.

LCD screens are cheaper, produce more accurate colors, and last longer than OLED screens. LCD screens also require a backlight and, thus, are brighter than OLED screens but consume more power to display black pixels. For a pixel to turn black in an LCD screen, it has to block the illumination from its backlight. In contrast, a pixel that is completely turned off in an OLED screen is black. As a result, an OLED screen displays deeper blacks since an LCD pixel can never perfectly block the backlight. Moreover, pixels that are pure black do not consume any power in an OLED screens, so a screen that contains mostly black pixels consumes significantly less battery than in an LCD screen.

The Moto 360 and the LG G watch use LCD screens while the LG G watch R uses an OLED screen. The Galaxy Gear Live and the ASUS ZenWatch also use a type of OLED screen called AMOLED.

This overview of screen types only scratches the surface, as there are several types of LCD screens and several types of OLED screens, each with its own benefits, detriments, and constraints. However, there are two display modes that watch faces should take into account.

Low-bit Ambient Mode

Certain types of screens, such as trans-reflective LCDs, are able to save additional battery by displaying only black and white pixels in ambient mode. While a screen in regular ambient mode can display any shade of gray, a screen in low-bit ambient mode should only display black and white pixels. If a watch requires low-bit ambient mode, Android Wear notifies a watch face through a callback method. To ensure that there are no shades of gray on the screen, watch faces in low-bit ambient mode should disable antialiasing. If antialiasing is enabled, drawing black and white content would likely result in many pixels that have different shades of gray, which is undesirable.

Note A screen is made up of a grid of squares called pixels, and drawing a diagonal line on a grid of squares leads to a jagged line that resembles a staircase. Antialiasing is the process modifying the color of each pixel surrounding a line to create the appearance that the line is smoother. Figure 7-2 shows a close-up of a bent line drawn with antialiasing disabled (left) and enabled (right).



Figure 7-2. A bent line drawn with antialiasing disabled (left) and enabled (right)

Burn-in Protection

Certain types of screens are susceptible to burn-in, where pixels begin to change colors as they are used more often. The prolonged use of certain pixels over others can result in ghost-like images that appear simply because those pixels have aged more than others. Android Wear attempts to minimize burn-in by periodically shifting the content of the entire screen by a few pixels at a time to avoid keeping the same pixels on for long periods. Minimizing burn-in is especially important in ambient mode, since watches are in ambient mode most of the day. However, periodically shifting pixels can only prevent burn-in if there are no large chunks of white content. Android Wear notifies watch faces if a screen is susceptible to burn-in by enabling burn-in protection mode.

Watch faces in this mode should avoid drawing large chunks of white pixels in ambient mode and instead draw shape outlines. Designs that require filled areas that cannot be substituted by outlines should use pixel patterns to avoid large areas of consecutive white pixels. If low-bit ambient mode is enabled in addition to burn-in protection, watch faces should disable antialiasing and draw only black or white pixels, as illustrated in Figure 7-3.

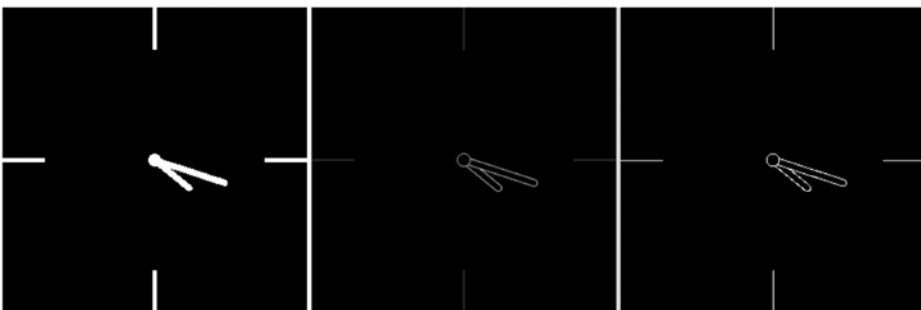


Figure 7-3. A watch face with low-bit ambient mode enabled (left), burn-in protection enabled (middle), and low-bit ambient mode and burn-in protection enabled (right)

Square and Round Screens

Watch faces should also take into account watches with square and round screens. Watch faces can either use a separate layout for each shape or a single layout that looks good on both shapes. In the former case, both layouts should use similar design elements and look like they belong to the same watch face even though they have different shapes. The examples in this chapter will illustrate both of these approaches.

In total, there are ten different combinations of configurations that watch faces should support, as illustrated by Figure 7-4.

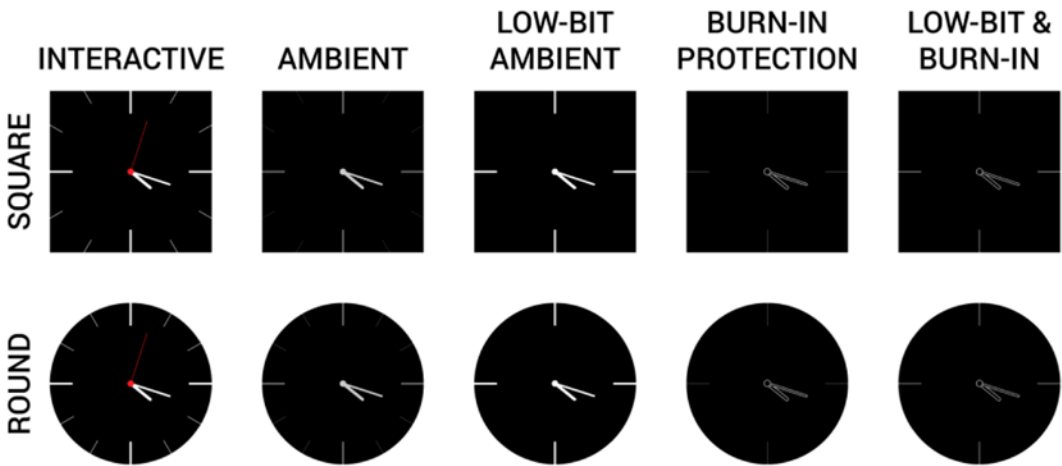


Figure 7-4. Ten combinations of screen shapes and modes that watch faces should support. Note that the tick marks in the right-most column are thin and white and that the hands of the clock are outlines drawn with antialiasing disabled

The Moto 360 and the Bottom Inset

The Moto 360 uses a round screen with a strange peculiarity: 30 pixels at the bottom of the screen are blank, as shown in Figure 7-5. This blank area is called the bottom inset or the chin and is used to hold electronics. Without the chin, the Moto 360 would not be able to have such a thin bezel.



Figure 7-5. The Moto 360 cannot display any content in the bottom-most 30 pixels

Round watch faces should still look good without the bottom most 30 pixels. Although some watch faces have a completely different design specifically for the Moto 360, most watch faces simply refrain from putting any essential information at the bottom. If, for instance, the chin only obscures a watch's tick marks or other non-essential components, the watch face can still be used on the Moto 360 without modification.

Interruption Filter

Android 5.0 devices, including Android Wear, support a setting called the interruption filter, which specifies what notifications should interrupt the user by vibrating or playing a notification sound. The interruption filter has three possible settings: show all, priority only, and none. If a watch face displays information related to notifications, such as the number of unread emails, it should hide this information when the user sets the interruption filter to none. Taking the interruption filter into account is a nice detail, but doing so is not anywhere near as important as taking into account all the previous considerations. Furthermore, watch faces that don't display any information related to notifications don't need to do anything in response to changes in the interruption filter.

System Indicators

Android Wear may overlay content on top of a watch face to indicate system status or display a preview of a notification. In particular, the system may display the content illustrated by Figure 7-6:

- The **status bar**, which displays icons that reflect the status of the system. In Figure 7-6, for example, the status bar contains three icons that indicate that the watch is in theater mode, disconnected from the phone, and charging, from left to right respectively. The status bar appears in both ambient and interactive modes.
- The **hotword indicator**, which appears as soon as the watch enters interactive mode and indicates that the user can say "Ok Google" to access the list of voice commands. The hotword indicator only appears in interactive mode.
- A **peek card**, which displays a preview of the notification that is at the top of the context stream. The peek card can appear in both ambient and interactive modes.

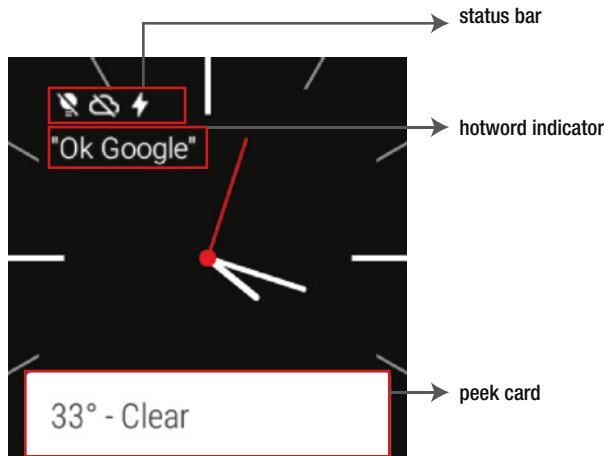


Figure 7-6. A watch face in interactive mode is displaying a status bar, a hotword indicator, and a peek card

When you are making a watch face, you are responsible for ensuring that

1. these indicators don't get in the way of your watch face and
2. your watch face doesn't get in the way of these indicators.

To do so, your watch face can set several properties to modify the appearance and position of these indicators. The next section describes these properties at a high level and shows how to change their values.

Configuring the Watch Face Style

The watch face style defines the parameters that control the appearance and position of the system indicators in a watch face. Programmatically, these parameters are contained in the `WatchFaceStyle` class, which can be created with the `WatchFaceStyle.Builder` class:

```
WatchFaceStyle watchFaceStyle = new WatchFaceStyle.Builder(Service watchFaceService)
    ...
    .build();
```

The builder takes a single parameter, which is the instance of the `CanvasWatchFaceService` that renders a watch face. The most important properties that can be set by the builder are enumerated below.

Note A watch face's `WatchFaceStyle` can be modified during run time. However, note that every parameter that isn't explicitly given to a new `WatchFaceStyle.Builder` has default values. If a watch face needs to change a single parameter of its `WatchFaceStyle`, it should either modify the existing `WatchFaceStyle` or create a new `WatchFaceStyle` that explicitly specifies any parameter that doesn't have a default value.

Status Bar Gravity

By default, the status bar is located at the top left of the watch face, but you can change its position by setting the status bar gravity. This property is the same gravity that controls the alignment of a view within its parent container. Set the status bar gravity with the `setStatusBarGravity(int gravity)` method:

```
new WatchFaceStyle.Builder(WatchFaceService.this)
    .setStatusBarGravity(gravity)
    .build();
```

where `gravity` is a flag that specifies horizontal and vertical alignment.

Specify horizontal alignment with one of the following flags:

- `Gravity.LEFT`
- `Gravity.CENTER_HORIZONTAL`
- `Gravity.RIGHT`

Specify vertical alignment with one of the following flags:

- `Gravity.TOP`
- `Gravity.CENTER_VERTICAL`
- `Gravity.BOTTOM`

You can combine horizontal and vertical alignments with a “bitwise or” such as `Gravity.RIGHT | Gravity.BOTTOM`. Note that `Gravity.CENTER` is equivalent to `Gravity.CENTER_HORIZONTAL | Gravity.CENTER_VERTICAL`. Figure 7-7 illustrates some possible values that the status bar gravity can take.

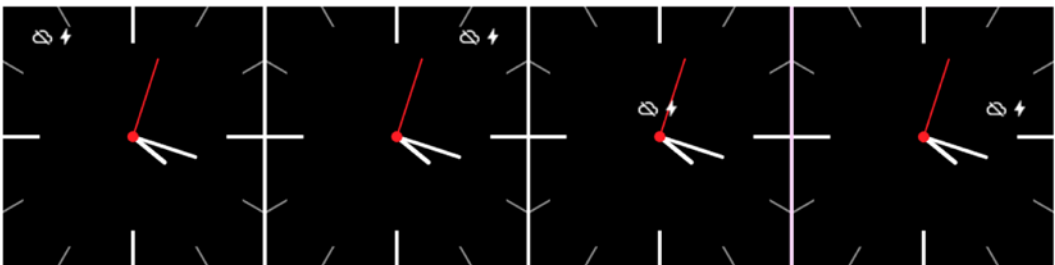


Figure 7-7. Examples of watch faces with different status bar gravities. From left to right, these watch faces use a status bar gravity of 1) `Gravity.LEFT`, 2) `Gravity.RIGHT`, 3) `Gravity.CENTER`, and 4) `Gravity.CENTER_HORIZONTAL | Gravity.RIGHT`. These values are not comprehensive as there are additional possible combinations

Card Peek Mode

By default, the peek card can occupy a variable amount of space at the bottom of the watch face. If a watch face can only accommodate short peek cards, their sizes can be constrained to a single line by changing the card peek mode with the `setCardPeekMode(int peekMode)` method:

```
new WatchFaceStyle.Builder(WatchFaceService.this)
    .setCardPeekMode(peekMode)
    .build();
```

where `peekMode` can take the values illustrated in Figure 7-8:

- `WatchFaceStyle.PEEK_MODE_VARIABLE` (default): the peek card's height varies according to its content.
- `WatchFaceStyle.PEEK_MODE_SHORT`: the peek card's height is constrained to a single line.



Figure 7-8. The card peek mode can take a value of `WatchFaceStyle.PEEK_MODE_VARIABLE` (left) and `WatchFaceStyle.PEEK_MODE_SHORT` (right)

Watch faces using the variable peek mode can ensure that a peek card does not block any information by placing important content on the top half of the screen or by dynamically moving content towards the top when a tall peek card appears. Watch faces without such a flexible UI should simply use the short peek mode and avoid placing important content close to the bottom of the screen.

The variable mode peek card at the left of Figure 7-8 covers the watch's hands at the bottom half of the screen and makes it impossible to tell time. This design should use a short mode peek card as shown on the right side of the figure.

Ambient Card Peek Mode

Peek cards in ambient mode are optional and can be revealed or hidden with the `setAmbientPeekMode(int ambientPeekMode)` method:

```
new WatchFaceStyle.Builder(WatchFaceService.this)
    .setAmbientPeekMode(WatchFaceStyle.AMBIENT_PEEK_MODE_HIDDEN)
    .build();
```

where `ambientPeekMode` can take the values illustrated by Figure 7-9:

- `WatchFaceStyle.AMBIENT_PEEK_MODE_VISIBLE` (default): the peek card is visible in ambient mode.
- `WatchFaceStyle.AMBIENT_PEEK_MODE_HIDDEN`: the peek card is hidden in ambient mode.

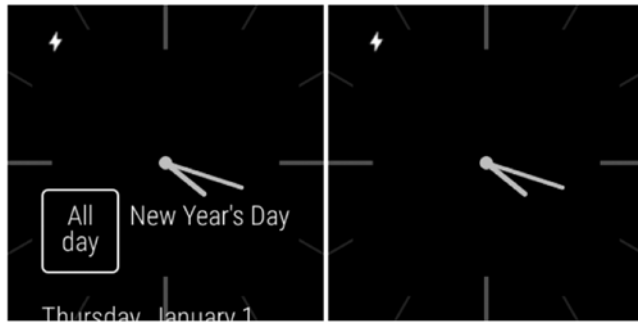


Figure 7-9. The ambient card peek mode can take a value of `WatchFaceStyle.AMBIENT_PEEK_MODE_VISIBLE` (left) or `WatchFaceStyle.AMBIENT_PEEK_MODE_HIDDEN` (right)

Note that visible peek cards in ambient mode do not have a background. If the watch face draws bright content near the bottom of the screen, it can interfere with the readability of the peek card. In this case, watch faces should draw a dark background behind peek cards to ensure that the peek card has enough contrast. We'll use this technique in an example later this chapter.

Peek Opacity Mode

Peek cards in interactive mode can have an opaque or a translucent background, as specified by the `setPeekOpacityMode(int peekOpacityMode)` method:

```
new WatchFaceStyle.Builder(WatchFaceService.this)
    .setPeekOpacityMode(peekOpacityMode)
    .build();
```

where `peekOpacityMode` can take the values illustrated by Figure 7-10:

- `WatchFaceStyle.PEEK_OPACITY_MODE_OPAQUE` (default)
- `WatchFaceStyle.PEEK_OPACITY_MODE_TRANSLUCENT`

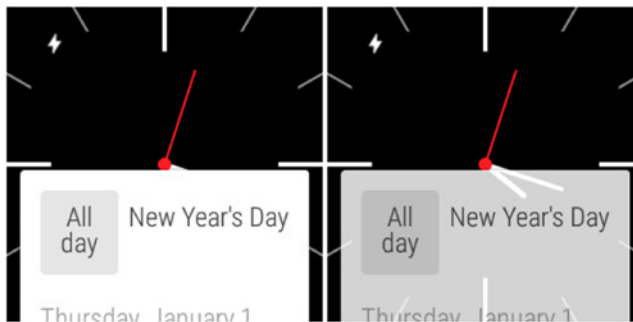


Figure 7-10. The peek opacity mode can have a value of `WatchFaceStyle.PEEK_OPACITY_MODE_OPAQUE` (left) or `WatchFaceStyle.PEEK_OPACITY_MODE_TRANSLUCENT` (right)

A translucent peek opacity mode reveals the watch face underneath the peek card at the expense of slightly reducing the peek card's contrast. Note that content under a translucent peek card is still difficult to see. In other words, avoid placing essential content underneath a translucent peek card.

Background Visibility

Cards in Android Wear can have backgrounds in addition to their main content. Since the peek card only displays a preview of a card, it does not necessarily display the card's background. The background visibility property determines when a peek card's background is displayed and can be specified with the `setBackgroundVisibility(int backgroundVisibility)` method:

```
new WatchFaceStyle.Builder(WatchFaceService.this)
    .setBackgroundVisibility(backgroundVisibility)
    .build();
```

where `backgroundVisibility` can take the values illustrated in Figure 7-11:

- `WatchFaceStyle.BACKGROUND_VISIBILITY_INTERRUPTIVE` (default): A peek card does not display its background in interactive mode.
- `WatchFaceStyle.BACKGROUND_VISIBILITY_PERSISTENT`: A peek card always displays its background in interactive mode, unless the user hides the peek card by swiping down on it. Note that the peek card's background obscures the watch face.

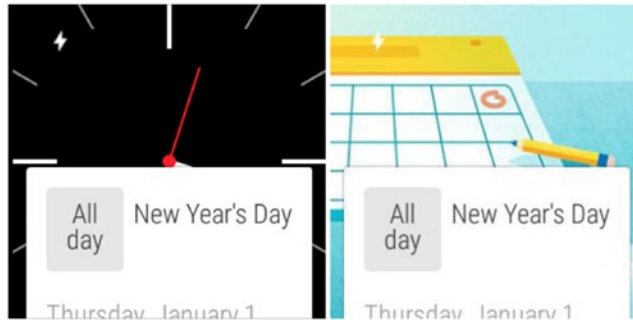


Figure 7-11. The background visibility can take a value of `WatchFaceStyle.BACKGROUND_VISIBILITY_INTERRUPTIVE` (left) or `WatchFaceStyle.BACKGROUND_VISIBILITY_PERSISTENT` (right)

Hotword Indicator Gravity

By default, the “Ok Google” hotword appears at the top left of the watch face. The hotword indicator gravity property allows a watch face to align the hotword to a different position and is specified by the `setHotwordIndicatorGravity(int hotwordIndicatorGravity)` method:

```
new WatchFaceStyle.Builder(WatchFaceService.this)
    .setHotwordIndicatorGravity(hotwordIndicatorGravity)
    .build();
```

where `hotwordIndicatorGravity` can take on the same values as the status bar gravity (see above) and the default value is `Gravity.LEFT`. Figure 7-12 illustrates some possible values that the hotword indicator gravity can take.

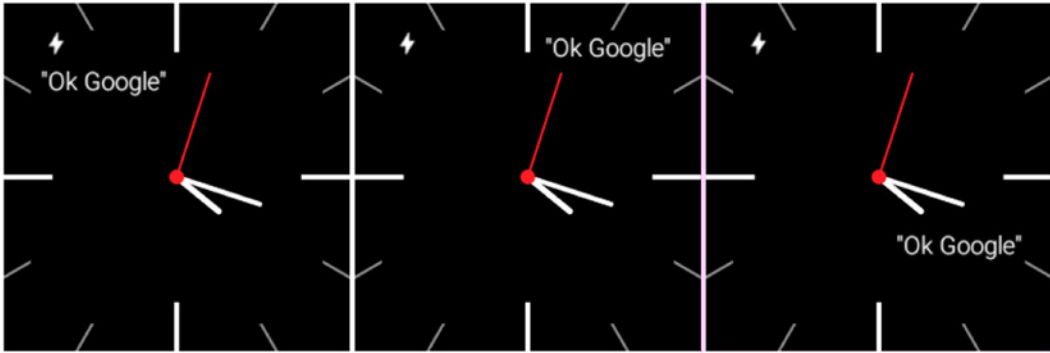


Figure 7-12. Examples of watch faces with different hotword indicator gravities. From right to left, these watch faces use a hotword indicator gravity of 1) Gravity.LEFT, 2) Gravity.RIGHT, and 3) Gravity.RIGHT | Gravity.BOTTOM

View Protection

The status bar and the hotword are displayed with white text. If a watch face has a light background, then these indicators can be difficult or impossible to read, as illustrated by the watch face on the left of Figure 7-13. View protection solves this problem by placing a dark translucent background behind an indicator, which increases its contrast. Specify view protection with the `setViewProtection(int viewProtection)` method:

```
new WatchFaceStyle.Builder(WatchFaceService.this)
    .setViewProtection(viewProtection)
    .build();
```

where `viewProtection` can have one of the values illustrated by Figure 7-13:

- 0 (default): disable view protection.
- `WatchFaceStyle.PROTECT_STATUS_BAR`: a dark translucent background is placed behind the status bar.
- `WatchFaceStyle.PROTECT_HOTWORD_INDICATOR`: a dark translucent background is placed behind the hotword indicator.
- `WatchFaceStyle.PROTECT_WHOLE_SCREEN`: a dark translucent background is placed behind the entire screen.

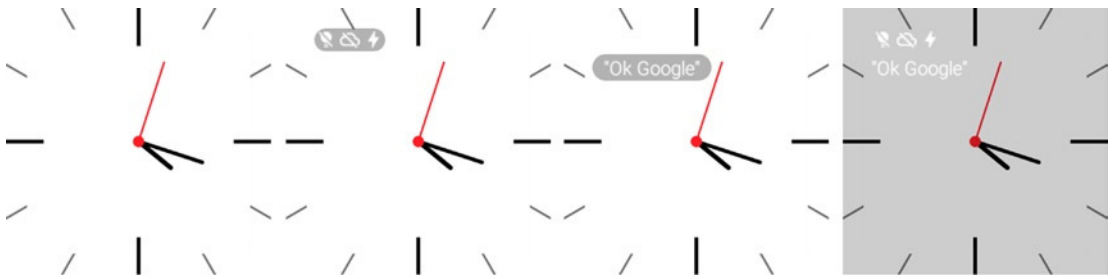


Figure 7-13. View protection increases the contrast between a view and its background. On the left, a watch face with a white background displays a status bar and a hotword indicator that cannot be seen because of low contrast. From left to right, the other watch faces have view protection set to 1) `WatchFaceStyle.PROTECT_STATUS_BAR`, 2) `WatchFaceStyle.PROTECT_HOTWORD_INDICATOR`, and 3) `WatchFaceStyle.PROTECT_WHOLE_SCREEN`

Show System UI Time

In addition to the indicators we have discussed so far, Android Wear can overlay the time on top of a watch face. This overlay is only necessary if the watch face does not already represent time. The watch face in Figure 7-14, for example, already represents the time as an analog clock, so the system UI time is redundant. However, displaying the time in both analog and digital forms can give users the best of both worlds, so this redundancy is not always bad. Enable the system UI time with the `setShowSystemUiTime(boolean showSystemUiTime)` method:

```
new WatchFaceStyle.Builder(WatchFaceService.this)
    .setShowSystemUiTime(showSystemUiTime)
    .build();
```

where `showSystemUiTime` is either true or false, as illustrated by Figure 7-14.



Figure 7-14. Showing the system UI time overlays the time in digital representation on top of the watch face

Building a Basic Watch Face

We've learned about watch faces, their different modes of operation, and the properties of their indicators. This section demonstrates how to implement the watch face used for all of the Figures shown so far. This watch face displays 12 tick marks that correspond to every hour, an hour hand, a minute hand, and a second hand that only appears in interactive mode (see Figure 7-4).

Note The source code for this example is located in the `wear` module of the `CustomWatchFaces` project. After running the module on an Android Wear device, a watch face called *Basic Watch Face* should be available. The watch face may take a few minutes to appear after the first time you run the module.

The watch face's implementation contains a lot of boilerplate code, so don't be intimidated by its quantity of code. The majority of this code can be re-used when you implement new watch faces.

Note Boilerplate is code that is used without change in any implementation of a certain type. In the case of watch faces, the code that determines the time, handles time zone changes, and handles configuration changes is boilerplate because it does not change among different watch faces.

Creating and Declaring a Watch Face

Watch faces are rendered from classes that extend `CanvasWatchFaceService`. We'll implement the basic watch face in a class called `BasicWatchFaceService`.

1. Create a new class called `BasicWatchFaceService` that extends `CanvasWatchFaceService`:

In `BasicWatchFaceService.java`:

```
public class BasicWatchFaceService extends CanvasWatchFaceService {
    @Override
    public CanvasWatchFaceService.Engine onCreateEngine() {
        return new Engine();
    }

    private class Engine extends CanvasWatchFaceService.Engine {
        @Override
        public void onDraw(Canvas canvas, Rect bounds) {
```

```

        super.onDraw(canvas, bounds);
        canvas.drawColor(Color.RED);
    }
}

```

The code that manages and draws a watch face belongs to the `Engine` class, which contains a variety of callbacks. For now, we'll simply draw a red background inside the `Engine` class, but we'll implement the real watch face later in this chapter.

2. Declare an xml resource that contains an empty wallpaper tag.

In res > xml > watch_face.xml:

```
<wallpaper xmlns:android="http://schemas.android.com/apk/res/android" />
```

3. Add the following permissions and features:

These permissions are required by all watch faces.

In `AndroidManifest.xml`:

```

<uses-permission android:name="com.google.android.permission.PROVIDE_BACKGROUND" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-feature android:name="android.hardware.type.watch" />

```

4. Declare the watch face itself.

In `AndroidManifest.xml`:

```

<service
    android:name=".BasicWatchFaceService"
    android:allowEmbedded="true"
    android:label="Basic Watch Face"
    android:permission="android.permission.BIND_WALLPAPER"
    android:taskAffinity="" >
    <meta-data
        android:name="android.service.wallpaper"
        android:resource="@xml/watch_face" />
    <meta-data
        android:name="com.google.android.wearable.watchface.preview"
        android:resource="@drawable/preview_basic" />
    <meta-data
        android:name="com.google.android.wearable.watchface.preview_circular"
        android:resource="@drawable/preview_basic_circular" />

    <intent-filter>
        <action android:name="android.service.wallpaper.WallpaperService" />
        <category android:name="com.google.android.wearable.watchface.category.WATCH_FACE" />
    </intent-filter>
</service>

```

The service declaration allows Android Wear to find a watch face's implementation and contains the following elements:

- `android:label` attribute: specifies the name of the watch face that will be shown to users when they are browsing available watch faces (see Figure 7-15).
- `android.service.wallpaper` meta-data: specifies an empty xml resource with nothing but a wallpaper tag. This meta-data is nothing but boilerplate that should be mindlessly copied (see step 2)
- `com.google.android.wearable.watchface.preview` and `com.google.android.wearable.watchface.preview_circular` meta-data: specify drawables that contains a preview image that will be shown to users when they browse available watch faces from a square and a round watch, respectively (see Figure 7-15).

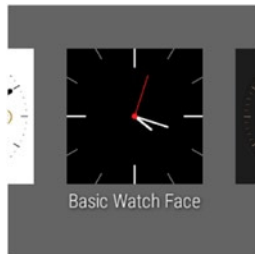


Figure 7-15. When users browse the list of available watch faces, a watch face's preview is generated from the information given by the manifest declaration

The drawables for the `preview` and the `preview_circular` meta-data should be screen captures of a watch face taken from a square and round watch, respectively. When beginning to implement a watch face, use random drawables for these resources as placeholders for the screen captures that you can take after completing the implementation.

At this point, you can run the *wear* module on an Android Wear device and see the watch face appear on the list of watch faces. If you select the watch face, you should see a static red screen regardless of whether you're in interactive or ambient mode. Next, we'll implement the Engine class with an actual watch face as opposed to a placeholder red screen.

Implementing the Engine Class

The Engine class is responsible for drawing a watch face, managing time zone changes, and figuring out when the watch is in interactive or ambient mode.

1. Declare constants and member variables.

In BasicWatchFaceService.java:

```
private class Engine extends CanvasWatchFaceService.Engine {
    private static final int MSG_UPDATE_WATCH_FACE = 0;
    private static final int INTERACTIVE_UPDATE_RATE_MS = 1000;
    private static final int DEGREES_PER_SECOND = 6;
    private static final int DEGREES_PER_MINUTE = 6;
    private static final int DEGREES_PER_HOUR = 30;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int REF_SIZE = 320;
    private Time mTime;
    private Paint mHourPaint, mMinutePaint, mSecondPaint;
    private Paint mMajorTickPaint, mMinorTickPaint, mBlackFillPaint, mWhiteFillPaint;
    private boolean mLowBitAmbient, mBurnInProtection, mAmbient;
    private boolean mRegisteredTimeZoneReceiver;
    private boolean mRound;
    private float mHourHandLength, mMinuteHandLength, mSecondHandLength;
    private float mMajorTickLength, mMinorTickLength, mMajorTickWidth;
    private float mClockHandCornerRadius;
    private int mPrevSize = -1;
    ...
}
```

2. Detect whether the device requires burn-in protection or is in low-bit ambient mode.

In BasicWatchFaceService.java:

```
@Override
public void onPropertiesChanged(Bundle properties) {
    super.onPropertiesChanged(properties);
    mLowBitAmbient = properties.getBoolean(PROPERTY_LOW_BIT_AMBIENT, false);
    mBurnInProtection = properties.getBoolean(PROPERTY_BURN_IN_PROTECTION, false);
}
}
```

3. Detect the size and position of the peek card. This information will be used while drawing the watch face in ambient mode.

In BasicWatchFaceService.java:

```
@Override
public void onPeekCardPositionUpdate(Rect rect) {
    super.onPeekCardPositionUpdate(rect);
    mPeekCardRect = rect;
}
}
```

4. Detect whether the device is square or round.

In `BasicWatchFaceService.java`:

```
@Override
public void onApplyWindowInsets(WindowInsets insets) {
    super.onApplyWindowInsets(insets);
    mRound = insets.isRound();
}
```

5. A “tick” is an event generated by Android Wear once every minute that notifies a watch face to update. In response to a tick event, invalidate the watch face to force the system to redraw the watch face.

In `BasicWatchFaceService.java`:

```
@Override
public void onTimeTick() {
    super.onTimeTick();
    invalidate();
}
```

6. A watch face should update at least every second in interactive mode, so the tick event occurs too slowly. To force the watch face to update every second, create a `Handler` that invalidates the watch face every second.

In `BasicWatchFaceService.java`:

```
private Handler mUpdateTimeHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        switch (msg.what) {
            case MSG_UPDATE_WATCH_FACE:
                invalidate();

                if(shouldContinueUpdatingTime()) {
                    long timeMs = System.currentTimeMillis();
                    long delayMs = INTERACTIVE_UPDATE_RATE_MS -
                        timeMs%INTERACTIVE_UPDATE_RATE_MS;

                    // delayMs is the delay up until the next even second (that is,
                    // the delay until the next multiple of 1000 milliseconds)
                    mUpdateTimeHandler.sendMessageDelayed(MSG_UPDATE_WATCH_FACE,
                        delayMs);
                }
                break;
        }
    }
};
```


7. When the watch face first becomes visible, attempt to start the Handler and register a time zone change listener. When the watch face is no longer visible, attempt to stop the Handler and unregister the time zone change listener.

The time zone change listener updates the time displayed on the watch face when a user changes time zone. The `updateTimer` method, which is called at the end of `onVisibilityChanged`, starts or stops a timer when the watch is in interactive or ambient mode, respectively. The timer invalidates the watch face every second so it can update fast enough to display a second hand.

In `BasicWatchFaceService.java`:

```
@Override
public void onVisibilityChanged(boolean visible) {
    super.onVisibilityChanged(visible);
    if(visible) {
        registerTimeZoneChangedReceiver();

        // update time zone in case it changed while watch was not visible
        mTime.clear(TimeZone.getDefault().getID());
        mTime.setToNow();
    } else {
        unregisterTimeZoneChangedReceiver();
    }

    updateTimer();
}
```

8. Implement the `updateTimer` method.

As explained in the previous step, this method makes sure that a “timer” is running only when the watch is interactive mode. The timer is really a Handler that repeatedly calls itself every second and invalidates the watch until it changes to ambient mode.

In `BasicWatchFaceService.java`:

```
private void updateTimer() {
    mUpdateTimeHandler.removeMessages(0);
    if(shouldContinueUpdatingTime()) {
        mUpdateTimeHandler.sendMessage(0);
    }
}

private boolean shouldContinueUpdatingTime() {
    return isVisible() && !isInAmbientMode();
}
```

9. The watch face should detect changes to the system's time zone and notify the `mTime` instance. The `mTime` member is an instance of the `Time` class and calculates the current hour, minute, and second based on the time zone and timestamp.

In `BasicWatchFaceService.java`:

```
private BroadcastReceiver mTimeZoneReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        mTime.clear(intent.getStringExtra("time-zone"));
        mTime.setToNow();
    }
};

private void registerTimeZoneChangedReceiver() {
    if(mRegisteredTimeZoneReceiver) {
        return;
    }
    mRegisteredTimeZoneReceiver = true;
    IntentFilter filter = new IntentFilter(Intent.ACTION_TIMEZONE_CHANGED);
    BasicWatchFaceService.this.registerReceiver(mTimeZoneReceiver, filter);
}

private void unregisterTimeZoneChangedReceiver() {
    if(!mRegisteredTimeZoneReceiver) {
        return;
    }
    mRegisteredTimeZoneReceiver = false;
    BasicWatchFaceService.this.unregisterReceiver(mTimeZoneReceiver);
}
```

10. When Engine is destroyed, stop Handler.

The Engine is destroyed when, for instance, a user switches to another watch face.

In `BasicWatchFaceService.java`:

```
@Override
public void onDestroy() {
    mUpdateTimeHandler.removeMessages(MSG_UPDATE_WATCH_FACE);
    super.onDestroy();
}
```

11. Detect when the watch face changes between interactive and ambient modes.

If low-bit ambient is enabled and the system is in ambient mode, disable antialiasing (by calling the helper method `setAntiAliasing(false)`). If burn-in protection is enabled and the system is in ambient mode, outline shapes instead of filling them.

In BasicWatchFaceService.java:

```
@Override
public void onAmbientModeChanged(boolean inAmbientMode) {
    super.onAmbientModeChanged(inAmbientMode);

    mAmbient = inAmbientMode;
    if(mLowBitAmbient) {
        setAntiAliasing(!mAmbient);
    }

    if(mBurnInProtection) {
        setBurnInProtection(mAmbient);
    }

    updateColors();
    invalidate();

    updateTimer();
}

private void setAntiAliasing(boolean antiAliasing) {
    mHourPaint.setAntiAlias(antiAliasing);
    mMinutePaint.setAntiAlias(antiAliasing);
    mSecondPaint.setAntiAlias(antiAliasing);
    mMajorTickPaint.setAntiAlias(antiAliasing);
    mMinorTickPaint.setAntiAlias(antiAliasing);
    mBlackFillPaint.setAntiAlias(antiAliasing);
    mWhiteFillPaint.setAntiAlias(antiAliasing);
}

private void setBurnInProtection(boolean enabled) {
    Paint.Style paintStyle = Paint.Style.FILL;
    if(enabled) {
        paintStyle = Paint.Style.STROKE;
    }

    mHourPaint.setStyle(paintStyle);
    mMinutePaint.setStyle(paintStyle);
    mWhiteFillPaint.setStyle(paintStyle);

    if(enabled) {
        mMajorTickPaint.setStrokeWidth(1f);
    } else {
        mMajorTickPaint.setStrokeWidth(mMajorTickWidth);
    }
}
```

12. In `onCreate`, initialize all of the `Paint` instances and enable antialiasing. Additionally, set the watch face to use a short card peek mode to avoid blocking the watch's hands.

These `Paint` objects are used to draw the watch face in the `onDraw` method, which we'll implement in the next section.

In `BasicWatchFaceService.java`:

```
@Override
public void onCreate(SurfaceHolder holder) {
    super.onCreate(holder);

    mHourPaint = new Paint();
    mHourPaint.setStrokeCap(Paint.Cap.ROUND);

    mMinutePaint = new Paint();
    mMinutePaint.setStrokeCap(Paint.Cap.ROUND);

    mSecondPaint = new Paint();
    mSecondPaint.setStrokeCap(Paint.Cap.ROUND);

    mMajorTickPaint = new Paint();
    mMinorTickPaint = new Paint();

    mBlackFillPaint = new Paint();
    mBlackFillPaint.setColor(Color.BLACK);

    mWhiteFillPaint = new Paint();

    setAntiAliasing(true);

    mTime = new Time();

    setWatchFaceStyle(new WatchFaceStyle.Builder(BasicWatchFaceService.this)
        .setCardPeekMode(WatchFaceStyle.PEEK_MODE_SHORT)
        .build());

    updateColors();
}
```

13. Implement the `updateColors` method, which was called in `onAmbientModeChanged`.

If the watch is in ambient mode and low-bit ambient is enabled, change all colors to white. Otherwise, update colors to their appropriate values depending on whether the watch is in ambient or interactive mode. Recall that a watch face should only draw black, white, and gray values when in ambient mode.

In `BasicWatchFaceService.java`:

```
private void updateColors() {
    boolean enableLowBitAmbient = mAmbient && mLowBitAmbient;
    if(enableLowBitAmbient) {
        // ambient mode with low-bit enabled
        mMajorTickPaint.setColor(Color.WHITE);
        mHourPaint.setColor(Color.WHITE);
        mMinutePaint.setColor(Color.WHITE);
        mWhiteFillPaint.setColor(Color.WHITE);
    } else if(mAmbient) {
        // ambient mode with low-bit disabled
        mHourPaint.setColor(Color.parseColor("#CCCCCC"));
        mMinutePaint.setColor(Color.parseColor("#CCCCCC"));
        mMajorTickPaint.setColor(Color.parseColor("#666666"));
        mMinorTickPaint.setColor(Color.parseColor("#333333"));
        mWhiteFillPaint.setColor(Color.parseColor("#CCCCCC"));
        mHourPaint.setColor(Color.parseColor("#CCCCCC"));
        mMinutePaint.setColor(Color.parseColor("#CCCCCC"));
    } else {
        // interactive mode
        mHourPaint.setColor(Color.WHITE);
        mMinutePaint.setColor(Color.WHITE);
        mSecondPaint.setColor(Color.parseColor("#FF1D25"));
        mMajorTickPaint.setColor(Color.WHITE);
        mMinorTickPaint.setColor(Color.GRAY);
        mWhiteFillPaint.setColor(Color.WHITE);
    }
}
```

Scaling the Watch Face to the Current Size

All of the widths and lengths of each element of the watch face—the watch hands and the tick marks—were obtained from a design that was previously sketched in a graphic design program.

Note Designing a watch face in code is a slow, tedious, and ineffective process. Instead, sketch designs for the watch face on paper and prototype them in graphic design software. If you are unfamiliar with design, either get help from a designer or learn the basics.

The original design was created on a 320x320 canvas. The size of the current watch could be different, and the `initDimensions` method scales all of the dimensions from the original design to the current watch size. This method assumes that the new watch size is a square or a circle (that is, that its width and height are the same).

In `BasicWatchFaceService.java`:

```
private void initDimensions(int size) {
    if(mPrevSize == size) {
        return;
    }
    mPrevSize = size;

    mHourHandLength = 50f / REF_SIZE * size;
    mMinuteHandLength = 81f / REF_SIZE * size;
    mSecondHandLength = 100f / REF_SIZE * size;

    mMajorTickLength = 45f / REF_SIZE * size;
    mMinorTickLength = 25f / REF_SIZE * size;

    mHourPaint.setStrokeWidth(1f);
    mMinutePaint.setStrokeWidth(1f);
    mSecondPaint.setStrokeWidth(Math.round(2f / REF_SIZE * size));

    mMajorTickWidth = Math.round(4f / REF_SIZE * size);
    mMajorTickPaint.setStrokeWidth(mMajorTickWidth);
    mMinorTickPaint.setStrokeWidth(Math.round(2f / REF_SIZE * size));

    mClockHandCornerRadius = Math.round(11f / REF_SIZE * size);
}
```

All that's left is to draw the watch face.

Drawing the Watch Face

The code we've implemented so far handles time zone changes, figures out when we are in ambient or interactive mode, and changes the colors of several `Paint` objects accordingly. Additionally, it handles low-bit ambient and burn-in protection if needed. The next step is to draw the watch face. The `Engine` class's `onDraw` method is responsible for drawing the watch face and is automatically called any time the watch face is invalidated.

1. When `onDraw` is called, start by finding the current time and determining the size of the watch. Then, figure out where the center of the watch face is and calculate the angle of each watch hand.

In `BasicWatchFaceService.java`:

```
@Override
public void onDraw(Canvas canvas, Rect bounds) {
    super.onDraw(canvas, bounds);

    mTime.setToNow();

    int width = bounds.width();
    int height = bounds.height();
```

```

// Find the center. Ignore the window insets so that, on round watches
// with a "chin", the watch face is centered on the entire screen, not
// just the usable portion.
float centerX = width / 2f;
float centerY = height / 2f;

// Compute rotations and lengths for the clock hands.
float secRot = (float)Math.toRadians(mTime.second * DEGREES_PER_SECOND);
float minutes = mTime.minute;
float minRot = (float)Math.toRadians(minutes * DEGREES_PER_MINUTE);
float hours = mTime.hour + minutes / MINUTES_PER_HOUR;
float hrRot = (float)Math.toRadians(hours * DEGREES_PER_HOUR );

// scale all the dimensions if needed
initDimensions(Math.min(width, height));
...

```

2. Draw an empty black background.

In the `onDraw` method of `BasicWatchFaceService.java`:

```
canvas.drawColor(Color.BLACK);
```

At this point, the result for interactive and ambient modes is just a black screen, as shown in Figure 7-16.



Figure 7-16. A watch face after drawing an empty black background

3. Draw the minor tick marks.

Draw lines across the entire screen at 30-degree intervals, and then cover up the middle of the watch with a square or circle, depending on the shape of the watch.

In the `onDraw` method of `BasicWatchFaceService.java`:

```

// draw the minor tick marks
if(!mAmbient || (!mBurnInProtection && !mLowBitAmbient)) {
    canvas.save();
    for (int i = 0; i < 12; ++i) {
        if (i % 3 != 0) {
            // draw a line wider than the screen to ensure it extends to the
            // diagonals of a square watch face
            canvas.drawLine(-100, centerY, width + 100, centerY, mMinorTickPaint);
        }
    }
}

```

```

        canvas.rotate(30f, centerX, centerY);
    }
    canvas.restore();

    if (mRound) {
        canvas.drawCircle(centerX, centerY, centerX - 32f, mBlackFillPaint);
    } else {
        canvas.drawRect(mMinorTickLength, mMinorTickLength, width - mMinorTickLength,
            height - mMinorTickLength, mBlackFillPaint);
    }
}

```

The result of drawing the lines is shown in Figure 7-17 and the result of drawing a square in the middle of the canvas is shown in Figure 7-18. Note that a circle would be drawn in the middle instead of a square if the current watch has a round screen.

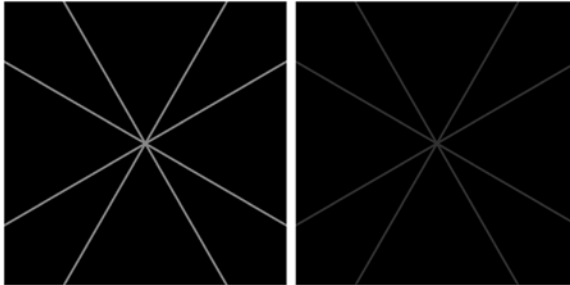


Figure 7-17. A watch face after drawing lines for the minor tick marks in interactive mode (left) and ambient mode (right)

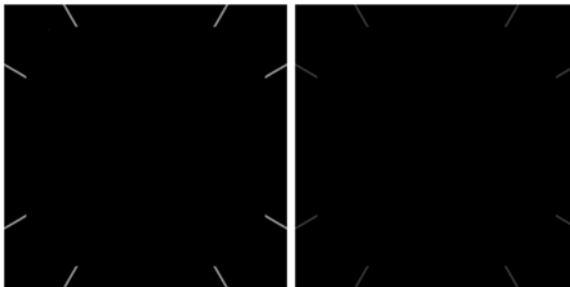


Figure 7-18. A square watch face after covering up the middle of the canvas with a black square in interactive mode (left) and ambient mode (right)

4. Draw the major tick marks.

The major tick marks are the ones directly at the top, bottom, left, and right of the watch face. Major tick marks are thicker than the minor tick marks, and they have a brighter color.

In the `onDraw` method of `BasicWatchFaceService.java`:

```
// draw major tick marks
canvas.drawLine(centerX, 0, centerX, mMajorTickLength, mMajorTickPaint);
canvas.drawLine(centerX, height - mMajorTickLength, centerX, height, mMajorTickPaint);
canvas.drawLine(0, centerY, mMajorTickLength, centerY, mMajorTickPaint);
canvas.drawLine(width - mMajorTickLength, centerY, width, centerY, mMajorTickPaint);
```

The result is shown in Figure 7-19.

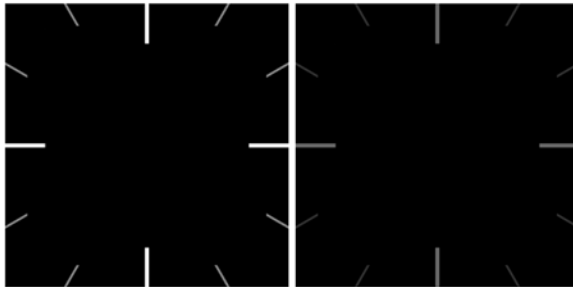


Figure 7-19. The watch face after drawing the major tick marks in interactive mode (left) and ambient mode (right)

5. Draw the hour and minute hands as narrow rectangles with slightly rounded corners.

In the `onDraw` method of `BasicWatchFaceService.java`:

```
// Draw the hour hand
float hrX = (float) Math.sin(hrRot) * mHourHandLength;
float hrY = (float) -Math.cos(hrRot) * mHourHandLength;
canvas.save();
canvas.rotate((float) Math.toDegrees(hrRot), centerX, centerY);
canvas.drawRoundRect(centerX - 3f, centerY - mHourHandLength, centerX + 3f, centerY,
    mClockHandCornerRadius, mClockHandCornerRadius, mHourPaint);
canvas.restore();

// Draw the minute hand on top of the hour hand
float minX = (float) Math.sin(minRot) * mMinuteHandLength;
float minY = (float) -Math.cos(minRot) * mMinuteHandLength;
canvas.save();
canvas.rotate((float) Math.toDegrees(minRot), centerX, centerY);
if(mBurnInProtection) {
    canvas.drawRoundRect(centerX - 3f, centerY - mMinuteHandLength, centerX + 3f,
        centerY, mClockHandCornerRadius, mClockHandCornerRadius, mBlackFillPaint);
}
canvas.drawRoundRect(centerX - 3f, centerY - mMinuteHandLength, centerX + 3f, centerY,
    mClockHandCornerRadius, mClockHandCornerRadius, mMinutePaint);
canvas.restore();
```

The result is shown in Figure 7-20.

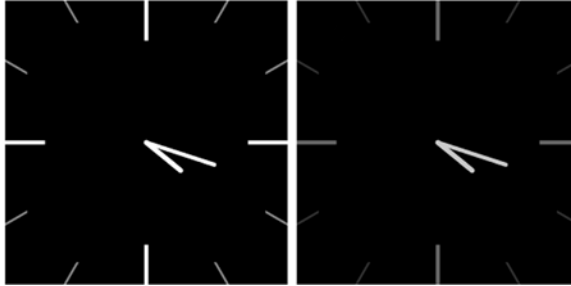


Figure 7-20. The watch face after drawing the hour and minute hands in interactive mode (left) and ambient mode (right)

6. Draw the second hand if the watch is in interactive mode and draw a small circle on top of the center of the watch hands.

Additionally, draw a black rectangle below the peek card in ambient mode to ensure its content has sufficient contrast.

In the `onDraw` method of `BasicWatchFaceService.java`:

```
if (mAmbient) {
    // draw center of watch hands
    if(mBurnInProtection) {
        canvas.drawCircle(centerX, centerY, 6f, mBlackFillPaint);
    }
    canvas.drawCircle(centerX, centerY, 6f, mWhiteFillPaint);

    // draw background for peek card
    if(mPeekCardRect != null) {
        canvas.drawRect(mPeekCardRect, mBlackFillPaint);
    }
} else {
    // Only draw the second hand in interactive mode.
    float secX = (float) Math.sin(secRot) * mSecondHandLength;
    float secY = (float) -Math.cos(secRot) * mSecondHandLength;
    canvas.drawLine(centerX, centerY, centerX + secX, centerY +
        secY, mSecondPaint);

    // draw center of watch hands
    canvas.drawCircle(centerX, centerY, 6f, mSecondPaint);
}
```

The final result is shown in Figure 7-21.

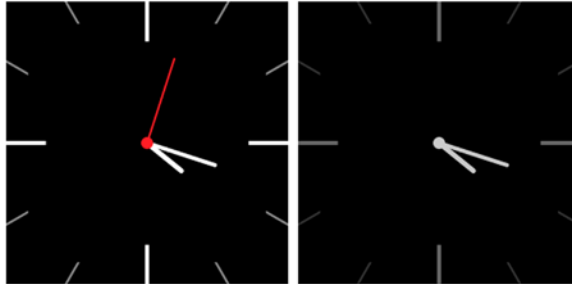


Figure 7-21. The watch face after drawing the second hand in interactive mode (left) and ambient mode (right)

At this point, run the watch face and verify that it works properly in interactive and ambient modes. To test low-bit ambient and burn-in protection modes, add the desired setting at the end of the `onPropertiesChanged` method. For instance, to test the watch face with both modes enabled, run the following code:

In `BasicWatchFaceService.java`:

```
@Override
public void onPropertiesChanged(Bundle properties) {
    super.onPropertiesChanged(properties);
    mLowBitAmbient = properties.getBoolean(PROPERTY_LOW_BIT_AMBIENT, false);
    mBurnInProtection = properties.getBoolean(PROPERTY_BURN_IN_PROTECTION, false);

    mLowBitAmbient = true;
    mBurnInProtection = true;
}
```

When building your own watch faces, be sure to test them with all four combinations of enabling and disabling both of these modes.

Building a Watch Face from Bitmaps

Drawing the entire watch face with lines, rectangles, and other shapes is only feasible for simple designs such as the previous example. To implement more complex designs, use images for the watch face's background and clock hands. This section will demonstrate how to use this technique to implement the watch face shown in Figure 7-22, which I've unoriginally called *Convergence*.

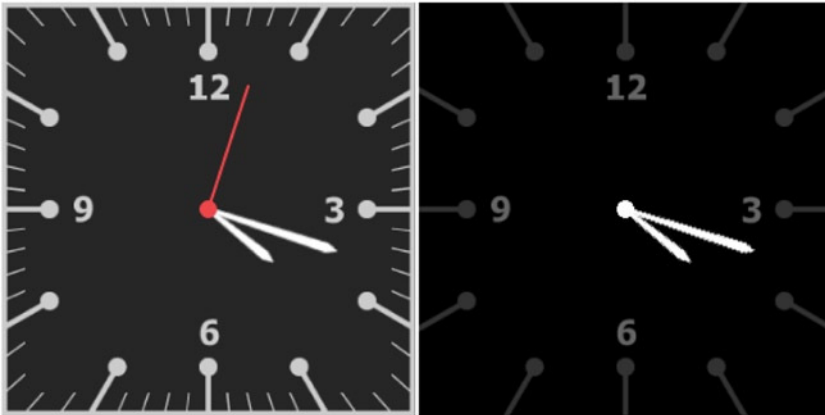


Figure 7-22. The “Convergence” watch face in interactive mode (left) and ambient mode (right)

Additionally, this example shows how to implement configuration activities, which allow users to configure the appearance of a watch face either from the watch or from a paired handheld.

Watch faces can have a configuration activity on the watch, on the handheld, or on both devices. To access the configuration activity on the watch, long press on the top of the current watch face to open the watch face selection screen. Watch faces with wearable configuration activities display a settings icon near the bottom of the screen, as shown in Figure 7-23. Tap on this icon to open the configuration activity.

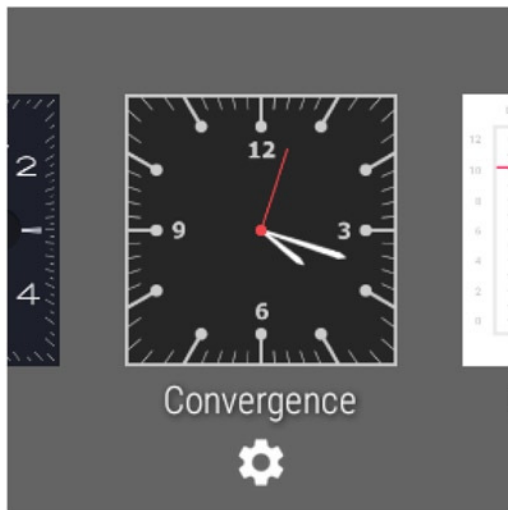


Figure 7-23. The Convergence watch face has a wearable configuration activity that can be accessed by tapping on the settings icon in the watch face selection screen

To access the configuration activity on a handheld device, open the Android Wear companion app. Watch faces with handheld configuration activities display a settings icon on top of the watch face preview (see Figure 7-24). Tap on this icon to open the configuration activity.

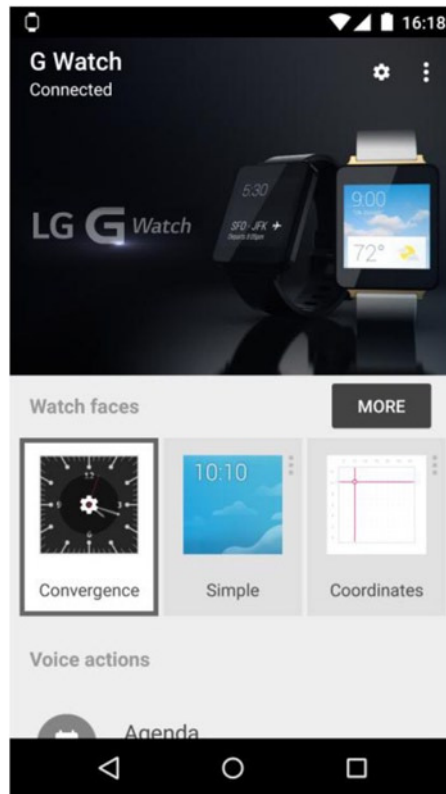


Figure 7-24. The convergence watch face has a handheld configuration activity that can be accessed by tapping on the settings icon in the Android Wear companion app

In the previous example, we drew the tick marks and the watch hands with lines and rectangles. Doing so was tedious and is only feasible for very simple designs since drawing arbitrary shapes cannot be easily accomplished. Instead of drawing the watch face in this example from primitive shapes, we'll draw each part of the watch as a bitmap.

Note The source code for this example is located in the `mobile` and `wear` module of the `CustomWatchFaces` project. After running the `wear` module on an Android Wear device, a watch face called *Convergence* should be available. The watch face may take a few minutes to appear after the first time you run the module. Then, after running the `mobile` module on a paired handheld device, the watch's configuration screen should be available on the companion app.

This example will use separate bitmaps to draw the background, hour hand, and minute hand. Recall that a watch face should take into account five states: interactive mode, ambient mode, lowbit ambient, burn-in protection, and lowbit ambient combined with burn-in protection. To handle these cases, every element of the watch face (that is, the background, hour hand, and minute hand) will have a separate bitmap for each state.

For example, the background of the watch can be drawn by any one of these files, illustrated by Figure 7-25:

- `cv_background_interactive_round`: the background of the watch face in interactive mode in round watches.
- `cv_background_interactive_square`: the background in interactive mode in square watches.
- `cv_background_ambient_round`: the background in ambient mode in round watches.
- `cv_background_ambient_square`: the background in ambient mode in square watches.
- `cv_background_lowbit_round`: the background in low-bit ambient mode in round watches.
- `cv_background_lowbit_square`: the background in low-bit ambient mode in square watches.
- `cv_background_burnin_round`: the background with burn-in protection enabled in ambient mode in round watches.
- `cv_background_burnin_square`: the background with burn-in protection enabled in ambient mode in square watches.
- `cv_background_lowbitburnin_round`: the background with low-bit and burn-in protection enabled in ambient mode in round watches.
- `cv_background_lowbitburnin_square`: the background with low-bit and burn-in protection enabled in ambient mode in square watches.

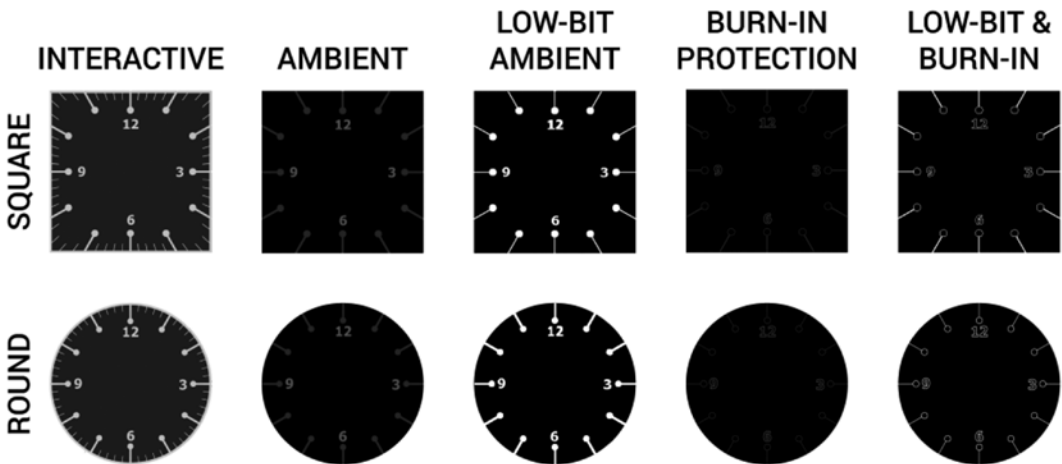


Figure 7-25. Every bitmap outlined above contains the background of the watch face for a particular watch shape in a particular mode

To manage all of these states for each watch face element, we'll write a class called `WatchFaceBitmapHolder`.

Encapsulating Bitmaps with WatchFaceBitmapHolder

As we saw in the previous section, an element of a watch face can use up to 10 different bitmaps to take into account all of the possible modes of operation of the watch. The `WatchFaceBitmapHolder` encapsulates the bitmaps of a particular watch face element for a particular shape. In other words, a `WatchFaceBitmapHolder` instance contains references to the five bitmaps that constitute a watch face element for a single watch shape. This object then returns the appropriate bitmap for the current state.

At a high level, here is how `WatchFaceBitmapHolder` works.

For a square watch, provide bitmap resources for the watch face element for 1) interactive mode, 2) ambient mode, 3) low-bit ambient mode, 4) burn-in protection, and 5) low-bit ambient + burn-in protection.

- Specify whether the watch face uses low-bit ambient or burn-in protection.
- Specify whether the watch face is currently in interactive mode.
- Call the `getBitmap` method to obtain the appropriate bitmap at any given time.
- Create another `WatchFaceBitmapHolder` for a round watch. When running, determine if the current watch is square or round and use the appropriate `WatchFaceBitmapHolder`.

The constructor of this class, which takes the resource identifiers of the bitmaps for every mode of operation, has the following signature:

```
public WatchFaceBitmapHolder(Context context, int interactiveResID, int ambientResId,
                             int lowBitResId, int burnInResId, int lowBitAndBurnInResId)
```

For instance, the background of the watch face with a round shape will be obtained from an instance of `WatchFaceBitmapHolder`:

```
mRoundBackgroundBitmapHolder = new WatchFaceBitmapHolder(ConvergenceWatchFaceService.this,
    R.drawable.cv_background_interactive_round,
    R.drawable.cv_background_ambient_round,
    R.drawable.cv_background_lowbit_round,
    R.drawable.cv_background_burnin_round,
    R.drawable.cv_background_lowbitburnin_round);
```

Now that we understand the purpose of this class, let's move on to its implementation.

1. Declare `WatchState` enum and **member** variables.

In the `wear` module's `WatchFaceBitmapHolder.java`:

```
public class WatchFaceBitmapHolder {
    public enum WatchState { INTERACTIVE, AMBIENT }
    private int mInteractiveId, mAmbientId, mLowBitAmbientId;
    private int mBurnInAmbientId, mLowBitAndBurnInId;
    private Bitmap mInteractiveScaledBitmap, mAmbientScaledBitmap;
```

```
private WatchState mWatchState;
private Context mContext;
private int mPrevWidth, mPrevHeight;
private int mRefWidth, mRefHeight;
...
```

2. Initialize member variables and obtain references to the resource IDs of the drawables that contain the bitmaps for the watch face element.

In the wear module's `WatchFaceBitmapHolder.java`:

```
public WatchFaceBitmapHolder(Context context, int interactiveResId, int ambientResId,
                             int lowBitResId, int burnInResId, int lowBitAndBurnInResId) {
    mContext = context;
    mInteractiveId = interactiveResId;
    mAmbientId = ambientResId;
    mLowBitAmbientId = lowBitResId;
    mBurnInAmbientId = burnInResId;
    mLowBitAndBurnInId = lowBitAndBurnInResId;
    mWatchState = WatchState.INTERACTIVE;
    mPrevWidth = -1;
    mPrevHeight = -1;
    mRefWidth = 320;
    mRefHeight = 320;
}
```

3. Create a setter that specifies the size of the watch for which the element was designed.

`WatchFaceBitmapHolder` scales a watch face element to the appropriate dimensions based on the reference size and the current size of the screen.

In the wear module's `WatchFaceBitmapHolder.java`:

```
public void setReferenceSize(int refWidth, int refHeight) {
    mRefWidth = refWidth;
    mRefHeight = refHeight;
}
```

4. Create a getter and a setter for the current state of the watch face (that is, interactive or ambient).

In the wear module's `WatchFaceBitmapHolder.java`:

```
public WatchState getWatchState() {
    return mWatchState;
}

public void setWatchState(WatchState watchState) {
    mWatchState = watchState;
}
```


5. Implement the `prepareBitmaps` method, which initializes the bitmaps and scales them to the appropriate dimensions.

This method selects the appropriate bitmap from all the available choices and scales it to the current watch's dimensions. Bitmaps are scaled according to the ratio of the size of the current watch to the size of the watch for which the element was designed. If a watch was designed for a 320x320 screen and the current watch has a 280x280 screen, the bitmaps are scaled by $280/320 = 0.875$.

This method only loads two bitmaps, one for interactive mode and another for ambient mode. The bitmap for ambient mode is selected according to the `lowbit` and `burn-in` protection settings.

In the `wear` module's `WatchFaceBitmapHolder.java`:

```
public void prepareBitmaps(int width, int height, boolean lowBitAmbient,
                          boolean burnInProtection) {
    if(mPrevWidth == width && mPrevHeight == height) {
        return;
    }

    mPrevWidth = width;
    mPrevHeight = height;

    int ambientId = mAmbientId;
    if(lowBitAmbient && burnInProtection) {
        ambientId = mLowBitAndBurnInId;
    } else if(lowBitAmbient && !burnInProtection) {
        ambientId = mLowBitAmbientId;
    } else if(!lowBitAmbient && burnInProtection) {
        ambientId = mBurnInAmbientId;
    }

    Resources res = mContext.getResources();
    Bitmap interactiveBitmap = ((BitmapDrawable)res.getDrawable(mInteractiveId)).getBitmap();
    Bitmap ambientBitmap = ((BitmapDrawable)res.getDrawable(ambientId)).getBitmap();

    int scaledWidth = interactiveBitmap.getWidth() * width / mRefWidth;
    int scaledHeight = interactiveBitmap.getHeight() * height / mRefHeight;

    // note: by convention, interactiveBitmap and ambientBitmap should have the same dimensions
    mInteractiveScaledBitmap = Bitmap.createScaledBitmap(interactiveBitmap,
        scaledWidth, scaledHeight, true);
    mAmbientScaledBitmap = Bitmap.createScaledBitmap(ambientBitmap, scaledWidth,
        scaledHeight, true);
}
```

6. Implement the `getBitmap` method.

After `prepareBitmaps` has been called, the class has loaded a bitmap for interactive mode and another for ambient mode. The `getBitmap` method will fetch the correct bitmap according to the mode defined by the `setWatchState` method.

In the "wear" module's `WatchFaceBitmapHolder.java`:

```
public Bitmap getBitmap() {
    if(mWatchState == WatchState.INTERACTIVE) {
        return mInteractiveScaledBitmap;
    } else {
        return mAmbientScaledBitmap;
    }
}
```

Data Layer Architecture

Before continuing, let's briefly discuss the architecture of this application, which has four primary components:

- `ConvergenceWatchFaceService`: renders the watch face
- `ConvergenceWearableConfigActivity`: provides a wearable configuration activity
- `ConvergenceConfigActivity`: resides on the handheld device and provides a handheld configuration activity
- `ConvergenceWearableListenerService`: receives messages from the `ConvergenceConfigActivity` and makes respective updates to the data API.

The data API is used to maintain the value of a Boolean called continuous sweep. When continuous sweep is enabled, the second hand on the watch moves with a continuous and steady motion. Continuous sweep, which can be enabled or disabled from either the handheld or the wearable, is stored as a `DataItem` that should only be updated from the watch. Updating a `DataItem` from both the handheld and the watch is error prone, so we tend to avoid it.

`ConvergenceWatchFaceService` obtains the value of continuous sweep with the data API.

`ConvergenceWearableConfigActivity` reads the initial value of continuous sweep from the data API and writes updated values.

`ConvergenceConfigActivity` reads the initial value of continuous sweep from the data API but does not write updates directly to it. Instead, this class sends the watch a message using the message API. This message is received by `ConvergenceWearableListenerService`, which in turn modifies the continuous sweep `DataItem`. Figure 7-26 illustrates this architecture.

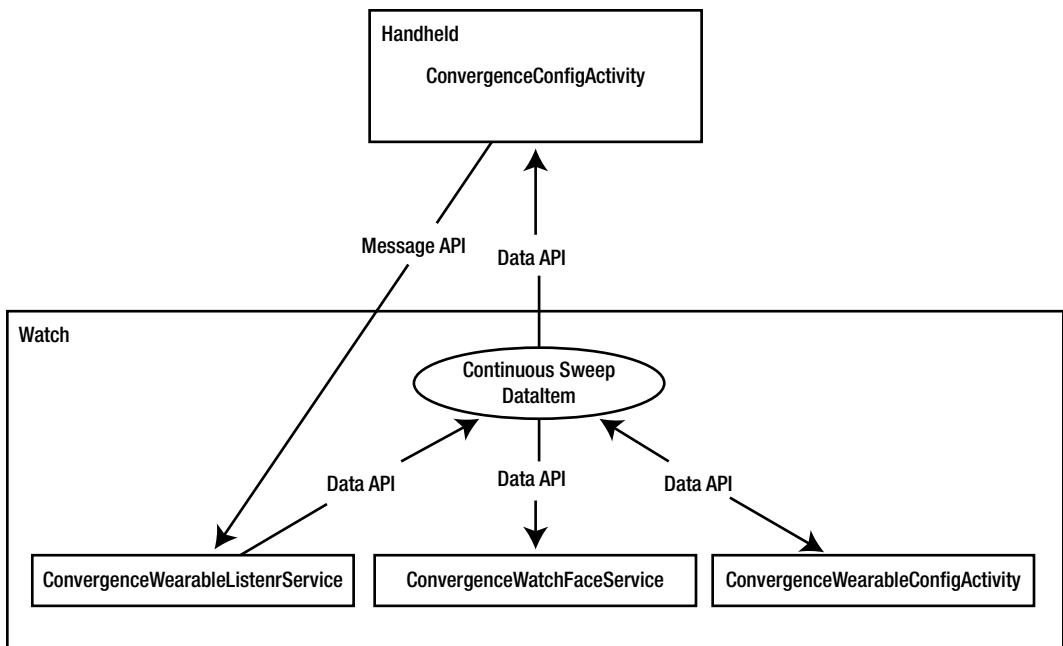


Figure 7-26. This diagram shows what classes update or read the value of continuous sweep. Note that a DataItem that contains the value of continuous sweep is only updated from the watch

Implementing ConvergenceUtil

The watch face in this example implements continuous sweep, which moves the second hand in a smooth and continuous motion while in interactive mode. Users can enable or disable continuous sweep from a configuration screen on the watch or on the handheld. When users enable or disable continuous sweep, the configuration activity uses the Wearable data layer API to update the watch face.

The `ConvergenceUtil` class contains methods that retrieve and insert the value of continuous sweep from and to the Wearable data layer API.

1. Declare constants.

The `PATH_CONTINUOUS_SWEEP` constant is the path of the `DataItem` that contains the value of continuous sweep. The `KEY_CONTINUOUS_SWEEP` constant is a key used to retrieve the value of continuous sweep from this `DataItem`.

In the wear module's `ConvergenceUtil.java`:

```

public class ConvergenceUtil {
    public static final String PATH_CONTINUOUS_SWEEP = "/convergence/sweep";
    public static final String KEY_CONTINUOUS_SWEEP = "sweep";
    ...
}
  
```

2. Implement the `putContinuousSweep` method, which inserts an updated value of continuous sweep into the data API.

In the wear module's `ConvergenceUtil.java`:

```
public static void putContinuousSweep(GoogleApiClient googleApiClient, boolean sweep) {
    PutDataMapRequest putDataMapRequest = PutDataMapRequest.create(PATH_CONTINUOUS_SWEEP);
    putDataMapRequest.getDataMap().putBoolean(KEY_CONTINUOUS_SWEEP, sweep);
    Wearable.DataApi.putDataItem(googleApiClient, putDataMapRequest.asPutDataRequest())
        .setResultCallback(new ResultCallback<DataApi.DataItemResult>() {
            @Override
            public void onResult(DataApi.DataItemResult dataItemResult) {
            }
        });
}
```

3. Implement the `extractContinuousSweep(DataEventBuffer)` method, which obtains the value of continuous sweep from a `DataEventBuffer`.

In the wear module's `ConvergenceUtil.java`:

```
public static boolean extractContinuousSweep(DataEventBuffer dataEvents) {
    final List<DataEvent> events = FreezableUtils
        .freezeIterable(dataEvents);

    for (DataEvent event : events) {
        if (event.getType() != DataEvent.TYPE_CHANGED) {
            continue;
        }

        DataItem dataItem = event.getDataItem();
        if (!dataItem.getUri().getPath().equals(ConvergenceUtil.PATH_CONTINUOUS_SWEEP)) {
            continue;
        }

        return extractContinuousSweep(dataItem);
    }

    return false;
}
```

4. Implement `extractContinuousSweep(DataItem)`, which extracts the value of continuous sweep from a `DataItem`.

This method defaults to false if the data item is null or does not contain a value of continuous sweep.

In the wear module's `ConvergenceUtil.java`:

```
private static boolean extractContinuousSweep(DataItem dataItem) {
    if(dataItem == null) {
        return false;
    }
}
```

```

DataMapItem dataMapItem = DataMapItem.fromDataItem(dataItem);
DataMap config = dataMapItem.getDataMap();
return config.getBoolean(KEY_CONTINUOUS_SWEEP, false);
}

```

5. Declare the `FetchContinuousSweepCallback` interface.

This interface provides a callback to fetch the value of continuous sweep asynchronously as we'll see in the next step.

In the wear module's `ConvergenceUtil.java`:

```

public interface FetchContinuousSweepCallback {
    void onContinuousSweepFetched(boolean continuousSweep);
}

```

6. Implement the `fetchContinuousSweep` method, which asynchronously retrieves the value of continuous sweep and passes it into a callback.

The value of continuous sweep resides in the Android Wear device, and the first step in retrieving its value is to find the local node ID.

In the "wear" module's `ConvergenceUtil.java`:

```

public static void fetchContinuousSweep(final GoogleApiClient client,
                                       final FetchContinuousSweepCallback callback) {
    Wearable.NodeApi.getLocalNode(client).setResultCallback(
        new ResultCallback<NodeApi.GetLocalNodeResult>() {
            @Override
            public void onResult(NodeApi.GetLocalNodeResult localNodeResult) {
                String localNode = localNodeResult.getNode().getId();
                fetchContinuousSweepDataItem(client, localNode, callback);
            }
        }
    );
}

```

7. Implement the `fetchContinuousSweepDataItem` method.

This method uses the local node ID found in the previous step to generate the URI of the `DataItem` that contains continuous sweep. It then uses the data API's `getDataItem` method to retrieve this `DataItem`. Finally, this method extracts the value of continuous sweep from the `DataItem` and passes it into the callback.

In the wear module's `ConvergenceUtil.java`:

```

private static void fetchContinuousSweepDataItem(GoogleApiClient client, String localNode,
                                                final FetchContinuousSweepCallback
callback) {
    Uri uri = new Uri.Builder()
        .scheme("wear")
        .path(PATH_CONTINUOUS_SWEEP)
        .authority(localNode)
        .build();
}

```

```

Wearable.DataApi.getDataItem(client, uri)
    .setResultCallback(new ResultCallback<DataApi.DataItemResult>() {
        @Override
        public void onResult(DataApi.DataItemResult dataItemResult) {
            boolean continuousSweep = extractContinuousSweep(dataItemResult.getDataItem());
            callback.onContinuousSweepFetched(continuousSweep);
        }
    });
}

```

The method we defined in `ConvergenceUtil` will be used throughout the rest of the chapter.

Implementing `ConvergenceWearableConfigActivity`

This activity allows users to enable or disable `continuous_sweep` directly from a watch. Recall that a user can open this activity by tapping on the *configuration* icon below a watch face in the watch face selection menu.

This activity contains a single checkbox that is checked when `continuous_sweep` is enabled. We limit the UI to a single checkbox for simplicity, but note that the UI is not ideal for Android Wear since it's inconsistent with the rest of the platform. Look at your watch's settings to see how to better handle selection.

1. Create a layout.

This layout only contains a checkbox titled `Continuous_sweep` at the center of the screen.

In `res > layout > activity_convergence_config.xml`:

```

<?xml version="1.0" encoding="utf-8"?>
<CheckBox
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/continuous_sweep"
    android:text="Continuous sweep"
    android:layout_gravity="center"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

```

2. Declare the activity as a watch face's wearable configuration screen.

In the `wear` module's `AndroidManifest.xml`:

The `intent-filter` in this declaration specifies that the activity is a wearable configuration screen for a watch face. We give the `intent-filter`'s action a custom name that is prefixed by the app's namespace. We'll use this name when declaring the watch face shortly.

```

<activity
    android:name=".ConvergenceWearableConfigActivity"
    android:label="Convergence Config">
    <intent-filter>
        <action android:name="
            "com.ocddevelopers.androidwearables.customwatchfaces.CONFIG_CONVERGENCE" />
    </intent-filter>

```

```

        <category android:name=
            "com.google.android.wearable.watchface.category.WEARABLE_CONFIGURATION" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>

```

3. Declare member variables.

In the wear module's `ConvergenceWearableConfigActivity.java`:

```

public class ConvergenceWearableConfigActivity extends Activity {
    private GoogleApiClient mGoogleApiClient;
    private CheckBox mContinuousSweep;
    ...

```

4. In `onCreate`, initialize `GoogleApiClient` and disable the continuous sweep checkbox.

We'll use `GoogleApiClient` to retrieve the initial value of continuous sweep from the data API, and we won't enable the checkbox until we connect to `GoogleApiClient` and retrieve this initial value.

In the wear module's `ConvergenceWearableConfigActivity.java`:

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_convergence_config);

    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addApi(Wearable.API)
        .addConnectionCallbacks(mConnectionCallbacks)
        .build();

    mContinuousSweep = (CheckBox) findViewById(R.id.continuous_sweep);
    mContinuousSweep.setOnCheckedChangeListener(mCheckedChangeListener);
    mContinuousSweep.setEnabled(false);
}

```

5. Connect to and disconnect from `GoogleApiClient` in the `onResume` and `onPause` methods, respectively.

In the wear module's `ConvergenceWearableConfigActivity.java`:

```

@Override
protected void onResume() {
    super.onResume();
    mGoogleApiClient.connect();
}

```

```

@Override
protected void onPause() {
    mGoogleApiClient.disconnect();
    super.onPause();
}

```

6. When the connection to `GoogleApiClient` is established, retrieve the initial value of `continuous_sweep`.

In the `wear` module's `ConvergenceWearableConfigActivity.java`:

```

private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks =
    new GoogleApiClient.ConnectionCallbacks() {
    @Override
    public void onConnected(Bundle bundle) {
        initializeContinuousSweep();
    }

    @Override
    public void onConnectionSuspended(int i) {
    }
};

```

7. To retrieve the initial value of `continuous_sweep` from the data API, use the `ConvergenceUtil.fetchContinuousSweep` method we wrote in the previous section.

After we retrieve the initial value of `continuous_sweep`, enable the checkbox so users can change its value.

In the `wear` module's `ConvergenceWearableConfigActivity.java`:

```

private void initializeContinuousSweep() {
    ConvergenceUtil.fetchContinuousSweep(mGoogleApiClient,
        new ConvergenceUtil.FetchContinuousSweepCallback() {
        @Override
        public void onContinuousSweepFetched(boolean continuousSweep) {
            mContinuousSweep.setChecked(continuousSweep);
            mContinuousSweep.setEnabled(true);
        }
    });
}

```


- When users change the value of `continuous_sweep` by tapping on the checkbox, update the value of `continuous_sweep` in the data API.

In the wear module's `ConvergenceWearableConfigActivity.java`:

```
private CompoundButton.OnCheckedChangeListener mCheckedChangeListener =
    new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        ConvergenceUtil.putContinuousSweep(mGoogleApiClient, isChecked);
    }
};
```

Now that we implemented a configuration activity for Android Wear, let's implement a configuration activity for the handheld.

Implementing ConvergenceConfigActivity

This activity allows users to change the value of `continuous_sweep` from a paired handheld device. Similar to the configuration activity on the wearable, this activity only contains a minimal UI that only contains a checkbox that is checked when `continuous_sweep` is enabled.

- Declare constants and member variables.

In the mobile module's `ConvergenceConfigActivity.java`:

```
public class ConvergenceConfigActivity extends ActionBarActivity {
    public static final String PATH_CONTINUOUS_SWEEP = "/convergence/sweep";
    public static final String KEY_CONTINUOUS_SWEEP = "sweep";
    private CheckBox mContinuousSweep;
    private GoogleApiClient mGoogleApiClient;
    private String mPeerId;
    ...
}
```

- Initialize `GoogleApiClient`, disable the `continuous_sweep` checkbox, and initialize `mPeerId`.

We'll use `GoogleApiClient` to retrieve the initial value of `continuous_sweep` from the data API, and we won't enable the checkbox until we connect to `GoogleApiClient` and retrieve this initial value. The `mPeerId` variable contains the node ID of the Android Wear device and is passed as an intent extra by the companion app.

In the mobile module's `ConvergenceConfigActivity.java`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_convergence_config);

    mContinuousSweep = (CheckBox) findViewById(R.id.continuous_sweep);
    mContinuousSweep.setEnabled(false);
    mContinuousSweep.setOnCheckedChangeListener(mCheckedChangeListener);
}
```

```

mGoogleApiClient = new GoogleApiClient.Builder(this)
    .addApi(Wearable.API)
    .addConnectionCallbacks(mConnectionCallbacks)
    .build();

mPeerId = getIntent().getStringExtra(WatchFaceCompanion.EXTRA_PEER_ID);
}

```

3. Connect to and disconnect from `GoogleApiClient` in the `onResume` and `onPause` methods, respectively.

In the mobile module's `ConvergenceConfigActivity.java`:

```

@Override
protected void onResume() {
    super.onResume();
    mGoogleApiClient.connect();
}

@Override
protected void onPause() {
    mGoogleApiClient.disconnect();
    super.onPause();
}

```

4. When the connection to `GoogleApiClient` is established, retrieve the value of continuous sweep.

If `mPeerId` is null, then the activity was started on a handheld device that is not paired to a watch. Although this situation should not happen under normal usage, we still guard against it just in case.

In the mobile module's `ConvergenceConfigActivity.java`:

```

private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks = new GoogleApiClient.
ConnectionCallbacks() {
    @Override
    public void onConnected(Bundle bundle) {

        if(mPeerId == null) {
            // unavailable
            Toast.makeText(getApplicationContext(), "not connected to wearable",
                Toast.LENGTH_SHORT).show();
            finish();
        } else {
            // get current checkbox state
            fetchContinuousSweep();
        }
    }
}

```

```

@Override
public void onConnectionSuspended(int i) {
}
};

```

5. Implement the `fetchContinuousSweep` method, which retrieves the value of continuous sweep from the data API.

Use the node ID of the watch (that is, `mPeerId`) to build a URI that we use to obtain the `DataItem` that contains the value of continuous sweep.

In the mobile module's `ConvergenceConfigActivity.java`:

```

private void fetchContinuousSweep() {
    Uri uri = new Uri.Builder()
        .scheme("wear")
        .path(PATH_CONTINUOUS_SWEEP)
        .authority(mPeerId)
        .build();

    Wearable.DataApi.getDataItem(mGoogleApiClient, uri).setResultCallback(
        new ResultCallback<DataApi.DataItemResult>() {
            @Override
            public void onResult(DataApi.DataItemResult dataItemResult) {
                DataItem dataItem = dataItemResult.getDataItem();
                initializeContinuousSweep(dataItem);
            }
        });
}

```

6. Implement the `initializeContinuousSweep` method, which extracts the value of continuous sweep from a `DataItem`, uses it to initialize the state of the checkbox, and enables the checkbox so users can change its value.

In the mobile module's `ConvergenceConfigActivity.java`:

```

private void initializeContinuousSweep(DataItem dataItem) {
    if(dataItem != null) {
        DataMapItem dataMapItem = DataMapItem.fromDataItem(dataItem);
        DataMap config = dataMapItem.getDataMap();
        boolean continuousSweep = config.getBoolean(KEY_CONTINUOUS_SWEEP);
        mContinuousSweep.setChecked(continuousSweep);
    }
    mContinuousSweep.setEnabled(true);
}

```

7. When users change the value of `continuous_sweep` by tapping on the checkbox, use the message API to notify the watch.

This activity does not modify the data API. Instead, it sends a message to the watch, and the watch modifies the value of `continuous_sweep` on the data API.

In the mobile module's `ConvergenceConfigActivity.java`:

```
private CompoundButton.OnCheckedChangeListener mCheckedChangeListener = new CompoundButton.
OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        sendConfigUpdateMessage(isChecked);
    }
};
```

8. Implement the `sendConfigUpdateMessage` method, which sends the updated value of `continuous_sweep` to the watch using the message API.

The state of `continuous_sweep` is stored as a byte array with a single item that is sent along with the message. If this item is greater than zero, `continuous_sweep` is enabled.

In the mobile module's `ConvergenceConfigActivity.java`:

```
private void sendConfigUpdateMessage(boolean continuousSweep) {
    if (mPeerId != null) {
        byte[] rawData = new byte[1];
        if(continuousSweep) {
            rawData[0] = 1;
        }
        Wearable.MessageApi.sendMessage(mGoogleApiClient, mPeerId,
            PATH_CONTINUOUS_SWEEP, rawData);
    }
}
```

Users can now modify the value of `continuous_sweep` from both their handhelds and their watches.

Implementing `ConvergenceWearableListenerService`

When users modify the value of `continuous_sweep` from their handhelds, the app uses the message API to notify the watch of the updated value. The `ConvergenceWearableService` class is responsible for receiving this message and updating the value of `continuous_sweep` on the data API. Recall that we use this architecture to ensure that only a single device updates a `DataItem`.

1. Declare `ConvergenceWearableListenerService`.

In the wear module's `AndroidManifest.xml`:

```
<service android:name=".ConvergenceWearableListenerService">
  <intent-filter>
    <action android:name="com.google.android.gms.wearable.BIND_LISTENER" />
  </intent-filter>
</service>
```

2. Declare and initialize `GoogleApiClient`.

In the wear module's `ConvergenceWearableListenerService.java`:

```
public class ConvergenceWearableListenerService extends WearableListenerService {
  private GoogleApiClient mGoogleApiClient;

  @Override
  public void onCreate() {
    super.onCreate();
    mGoogleApiClient = new GoogleApiClient.Builder(this)
      .addApi(Wearable.API)
      .build();
  }
  ...
}
```

3. When a message is received, connect to `GoogleApiClient` synchronously, retrieve the value of `continuous sweep` from the message's byte array, and update the `DataItem` that contains `continuous sweep`.

In the wear module's `ConvergenceWearableListenerService.java`:

```
@Override
public void onMessageReceived(MessageEvent messageEvent) {
  super.onMessageReceived(messageEvent);
  if (!messageEvent.getPath().equals(ConvergenceUtil.PATH_CONTINUOUS_SWEEP)) {
    return;
  }

  ConnectionResult connectionResult = mGoogleApiClient.blockingConnect(10, TimeUnit.SECONDS);
  if (!connectionResult.isSuccess()) {
    return;
  }

  boolean continuousSweep = messageEvent.getData()[0] > 0;
  ConvergenceUtil.putContinuousSweep(mGoogleApiClient, continuousSweep);

  mGoogleApiClient.disconnect();
}
```

Connecting to `GoogleApiClient` synchronously is possible since the `onMessageReceived` method is not called on the UI thread.

Implementing ConvergenceWatchFaceService

Use the previous example watch face, `BasicWatchFaceService`, as a template for this implementation. The majority of the code from that example is also applicable here.

Start by declaring the watch face in the manifest as follows.

In the wear module's `AndroidManifest.xml`:

```
<service
    android:name=".ConvergenceWatchFaceService"
    android:allowEmbedded="true"
    android:label="Convergence"
    android:permission="android.permission.BIND_WALLPAPER"
    android:taskAffinity="" >
    <meta-data
        android:name="android.service.wallpaper"
        android:resource="@xml/watch_face" />
    <meta-data
        android:name="com.google.android.wearable.watchface.preview"
        android:resource="@drawable/preview_convergence" />
    <meta-data
        android:name="com.google.android.wearable.watchface.preview_circular"
        android:resource="@drawable/preview_convergence_circular" />

    <meta-data
        android:name="com.google.android.wearable.watchface.companionConfigurationAction"
        android:value="com.ocddevelopers.androidwearables.customwatchfaces.CONFIG_CONVERGENCE"/>

    <meta-data
        android:name="com.google.android.wearable.watchface.wearableConfigurationAction"
        android:value="com.ocddevelopers.androidwearables.customwatchfaces.CONFIG_CONVERGENCE"/>

    <intent-filter>
        <action android:name="android.service.wallpaper.WallpaperService" />
        <category android:name="com.google.android.wearable.watchface.category.WATCH_FACE" />
    </intent-filter>
</service>
```

Note that this declaration contains meta-data that associates the watch face with a handheld and a wearable configuration activity. The value of these meta-data should match the name of the actions of the intent-filters in the configuration activities. See the declarations of `ConvergenceWearableConfigActivity` in the wear module and `ConvergenceConfigActivity` in the mobile module.

Initializing the Paints and Bitmaps

1. Declare constants and member variables.

In the wear module's `ConvergenceWatchFaceService.java`:

```
private class Engine extends CanvasWatchFaceService.Engine {
    private static final int MSG_UPDATE_WATCH_FACE = 0;
    private static final int INTERACTIVE_UPDATE_RATE_MS = 1000;
    private static final int DEGREES_PER_SECOND = 6;
    private static final int DEGREES_PER_MINUTE = 6;
    private static final int DEGREES_PER_HOUR = 30;
    private static final int MINUTES_PER_HOUR = 60;
    private static final int REF_SIZE = 320;
    private Time mTime;
    private Paint mSecondPaint, mBlackFillPaint, mWhiteFillPaint;
    private Paint mBitmapPaint, mDatePaint, mTimePaint;
    private boolean mLowBitAmbient, mBurnInProtection, mAmbient;
    private boolean mRegisteredTimeZoneReceiver;
    private boolean mRound;
    private float mSecondHandLength;
    private Rect mPeekCardRect;
    private WatchFaceBitmapHolder mRoundBackgroundBitmapHolder,
mSquareBackgroundBitmapHolder;
    private WatchFaceBitmapHolder mHourHandBitmapHolder, mMinuteHandBitmapHolder;
    private GoogleApiClient mGoogleApiClient;
    private boolean mContinuousSweep;
    ...
}
```

2. In `onCreate`, initialize `WatchFaceBitmapHolder`s for the watch face's background, hour hand, and minute hand. The second hand does not require `WatchFaceBitmapHolder`s since we still draw it based on primitive shapes (namely, a line and a circle).

In the wear module's `ConvergenceWatchFaceService.java`:

```
@Override
public void onCreate(SurfaceHolder holder) {
    super.onCreate(holder);

    mBitmapPaint = new Paint();

    mSecondPaint = new Paint();
    mSecondPaint.setStrokeCap(Paint.Cap.ROUND);
    mSecondPaint.setColor(Color.parseColor("#ef464c"));
    mSecondPaint.setStrokeWidth(2f);

    mBlackFillPaint = new Paint();
    mBlackFillPaint.setColor(Color.BLACK);

    mWhiteFillPaint = new Paint();
    mWhiteFillPaint.setColor(Color.WHITE);
}
```

```
mDatePaint = new Paint();
mDatePaint.setTextAlign(Paint.Align.CENTER);
mDatePaint.setTypeface(Typeface.create("sans-serif-light", Typeface.NORMAL));
mDatePaint.setColor(Color.parseColor("#d7dada"));
mDatePaint.setTextSize(18f);
mDatePaint.setShadowLayer(1f, 4f, 4f, Color.BLACK);

mTimePaint = new Paint();
mTimePaint.setTextAlign(Paint.Align.CENTER);
mTimePaint.setTypeface(Typeface.create("sans-serif-medium", Typeface.NORMAL));
mTimePaint.setColor(Color.parseColor("#d7dada"));
mTimePaint.setTextSize(26f);
mTimePaint.setShadowLayer(1f, 2f, 2f, Color.BLACK);

mRoundBackgroundBitmapHolder = new WatchFaceBitmapHolder(ConvergenceWatchFaceService.this,
    R.drawable.cv_background_interactive_round,
    R.drawable.cv_background_ambient_round,
    R.drawable.cv_background_lowbit_round,
    R.drawable.cv_background_burnin_round,
    R.drawable.cv_background_lowbitburnin_round);

mSquareBackgroundBitmapHolder = new WatchFaceBitmapHolder(ConvergenceWatchFaceService.this,
    R.drawable.cv_background_interactive_square,
    R.drawable.cv_background_ambient_square,
    R.drawable.cv_background_lowbit_square,
    R.drawable.cv_background_burnin_square,
    R.drawable.cv_background_lowbitburnin_square);

mHourHandBitmapHolder = new WatchFaceBitmapHolder(ConvergenceWatchFaceService.this,
    R.drawable.cv_hour_hand_filled,
    R.drawable.cv_hour_hand_filled,
    R.drawable.cv_hour_hand_lowbit,
    R.drawable.cv_hour_hand_burnin,
    R.drawable.cv_hour_hand_lowbitburnin);

mMinuteHandBitmapHolder = new WatchFaceBitmapHolder(ConvergenceWatchFaceService.this,
    R.drawable.cv_minute_hand_filled,
    R.drawable.cv_minute_hand_filled,
    R.drawable.cv_minute_hand_lowbit,
    R.drawable.cv_minute_hand_burnin,
    R.drawable.cv_minute_hand_lowbitburnin);

setAntiAliasing(true);
mTime = new Time();

setWatchFaceStyle(new WatchFaceStyle.Builder(ConvergenceWatchFaceService.this)
    .setCardPeekMode(WatchFaceStyle.PEEK_MODE_SHORT)
    .build());

mGoogleApiClient = new GoogleApiClient.Builder(ConvergenceWatchFaceService.this)
    .addApi(Wearable.API)
    .addConnectionCallbacks(mConnectionCallbacks)
    .build();
}
```


3. Additional instances of `Paint` need antialiasing toggled in low-bit ambient mode.

In the `wear` module's `ConvergenceWatchFaceService.java`:

```
private void setAntiAliasing(boolean antiAliasing) {
    mBitmapPaint.setAntiAlias(antiAliasing);
    mBitmapPaint.setFilterBitmap(antiAliasing);
    mSecondPaint.setAntiAlias(antiAliasing);
    mWhiteFillPaint.setAntiAlias(antiAliasing);
    mDatePaint.setAntiAlias(antiAliasing);
    mTimePaint.setAntiAlias(antiAliasing);
}
```

Retrieving Continuous Sweep with the Data API

The watch face needs to retrieve the value of `continuous_sweep` so it can draw the second hand accordingly.

1. When the connection to `GoogleApiClient` is established, retrieve the initial value of `continuous_sweep` and register a `DataListener` with the data API to receive updates to the `continuous_sweep`.

In particular, the `DataListener` is triggered when users change the value of `continuous_sweep` from one of the configuration activities.

In the `wear` module's `ConvergenceWatchFaceService.java`:

```
private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks =
    new GoogleApiClient.ConnectionCallbacks() {
        @Override
        public void onConnected(Bundle bundle) {
            initializeContinuousSweep();
            Wearable.DataApi.addListener(mGoogleApiClient, mDataListener);
        };

        @Override
        public void onConnectionSuspended(int i) {
        }
    };
```

2. Retrieve the initial value of `continuous_sweep` using the `ConvergenceUtil.fetchContinuousSweep` method.

In the `wear` module's `ConvergenceWatchFaceService.java`:

```
private void initializeContinuousSweep() {
    ConvergenceUtil.fetchContinuousSweep(mGoogleApiClient,
        new ConvergenceUtil.FetchContinuousSweepCallback() {
            @Override
```

```

        public void onContinuousSweepFetched(final boolean continuousSweep) {
            updateContinuousSweep(continuousSweep);
        }
    });
}

```

3. When a new value of `continuous sweep` is received, call the `updateTimer` method and invalidate the watch face.

The `updateTimer` method starts or stops the timer that invalidates the watch face every second. This timer is only needed when `continuous sweep` is disabled. When `continuous sweep` is enabled, the `onDraw` method takes care of invalidating the watch face, as we'll see shortly.

In the `wear` module's `ConvergenceWatchFaceService.java`:

```

private void updateContinuousSweep(final boolean continuousSweep) {
    mContinuousSweep = continuousSweep;
    updateTimer();
    invalidate();
}

```

4. Modify the `updateTimer` method to only start the timer when `continuous sweep` is disabled.

In the `wear` module's `ConvergenceWatchFaceService.java`:

```

private void updateTimer() {
    mUpdateTimeHandler.removeMessages(MSG_UPDATE_WATCH_FACE);
    if(shouldContinueUpdatingTime() && !mContinuousSweep) {
        mUpdateTimeHandler.sendMessage(MSG_UPDATE_WATCH_FACE);
    }
}

```

5. When the value of `continuous sweep` changes at run time, call the `updateContinuousSweep` method to update the watch face accordingly.

In the `wear` module's `ConvergenceWatchFaceService.java`:

```

private DataApi.DataListener mDataListener = new DataApi.DataListener() {
    @Override
    public void onDataChanged(DataEventBuffer dataEvents) {
        updateContinuousSweep(ConvergenceUtil.extractContinuousSweep(dataEvents));
    }
};

```

6. Modify the `onVisibilityChanged` method to connect to or disconnect from `GoogleApiClient` when the watch face becomes visible or is no longer visible, respectively.

In the wear module's `ConvergenceWatchFaceService.java`:

```
@Override
public void onVisibilityChanged(boolean visible) {
    super.onVisibilityChanged(visible);
    if(visible) {
        registerTimeZoneChangedReceiver();

        // update time zone in case it changed while watch was not visible
        mTime.clear(TimeZone.getDefault().getID());
        mTime.setToNow();

        mGoogleApiClient.connect();
    } else {
        unregisterTimeZoneChangedReceiver();

        if (mGoogleApiClient != null && mGoogleApiClient.isConnected()) {
            Wearable.DataApi.removeListener(mGoogleApiClient, mDataListener);
            mGoogleApiClient.disconnect();
        }
    }

    updateTimer();
}
```

Drawing the Watch Face

In this section, we'll implement the watch face's `onDraw` method, which is responsible for drawing the content of the watch face.

1. Declare the activity as a watch face's handheld configuration screen.

The intent-filter in this declaration specifies that the activity is a handheld configuration screen for a watch face. We give the intent-filter's action a custom name that is prefixed by the app's namespace. We'll use this name when declaring the watch face shortly.

In the mobile module's `AndroidManifest.xml`:

```
<activity
    android:name=".ConvergenceConfigActivity"
    android:label="Convergence Configuration">
    <intent-filter>
        <action android:name=
            "com.ocddevelopers.androidwearables.customwatchfaces.CONFIG_CONVERGENCE" />
        <category android:name=
            "com.google.android.wearable.watchface.category.COMPANION_CONFIGURATION" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

2. Obtain the current time, including the number of milliseconds.

If continuous sweep is disabled, we don't care about the number of milliseconds so we leave it at zero.

In the "wear" module's `ConvergenceWatchFaceService.java`:

```
@Override
public void onDraw(Canvas canvas, Rect bounds) {
    super.onDraw(canvas, bounds);

    long now = System.currentTimeMillis();
    mTime.set(now);
    int milliseconds = 0;
    if(mContinuousSweep) {
        milliseconds = (int) (now % 1000);
    }
    ...
}
```

3. Obtain the dimensions of the current watch face and find the coordinates of its center.

In the "wear" module's `ConvergenceWatchFaceService.java`:

```
int width = bounds.width();
int height = bounds.height();

// Find the center. Ignore the window insets so that, on round watches
// with a "chin", the watch face is centered on the entire screen, not
// just the usable portion.
float centerX = width / 2f;
float centerY = height / 2f;
```

4. Calculate the angle of each watch hand based on the current time.

In the "wear" module's `ConvergenceWatchFaceService.java`:

```
// Compute rotations and lengths for the clock hands.
float decimalSeconds = mTime.second + milliseconds/1000f;
float secRotRad = (float)Math.toRadians(decimalSeconds * DEGREES_PER_SECOND);
float minutes = mTime.minute;
float minRot = minutes * DEGREES_PER_MINUTE;
float hours = mTime.hour + minutes / MINUTES_PER_HOUR;
float hrRot = hours * DEGREES_PER_HOUR;
```

5. Call `prepareBitmaps` for every instance of `WatchFaceBitmapHolder` to ensure that the bitmaps are scaled to the appropriate dimensions of the current watch face.

In the "wear" module's `ConvergenceWatchFaceService.java`:

```
mRoundBackgroundBitmapHolder.prepareBitmaps(width, height,
    mLowBitAmbient, mBurnInProtection);
mSquareBackgroundBitmapHolder.prepareBitmaps(width, height,
    mLowBitAmbient, mBurnInProtection);
mHourHandBitmapHolder.prepareBitmaps(width, height,
    mLowBitAmbient, mBurnInProtection);
mMinuteHandBitmapHolder.prepareBitmaps(width, height,
    mLowBitAmbient, mBurnInProtection);
```

6. Notify these instances when the watch changes between interactive and ambient mode.

In the "wear" module's `ConvergenceWatchFaceService.java`:

```
WatchFaceBitmapHolder.WatchState state = (mAmbient)?
    WatchFaceBitmapHolder.WatchState.AMBIENT :
    WatchFaceBitmapHolder.WatchState.INTERACTIVE;
mRoundBackgroundBitmapHolder.setWatchState(state);
mSquareBackgroundBitmapHolder.setWatchState(state);
mHourHandBitmapHolder.setWatchState(state);
mMinuteHandBitmapHolder.setWatchState(state);
...
```

7. Draw the background, hour, and minute hands.

In the wear module's `ConvergenceWatchFaceService.java`:

```
...
// draw background
Bitmap background = (mRound)? mRoundBackgroundBitmapHolder.getBitmap() :
    mSquareBackgroundBitmapHolder.getBitmap();
canvas.drawBitmap(background, 0f, 0f, mBitmapPaint);

// draw hour hand
Bitmap hourHand = mHourHandBitmapHolder.getBitmap();
canvas.save();
canvas.rotate(hrRot, centerX, centerY);
canvas.drawBitmap(hourHand, centerX-hourHand.getWidth()/2f,
    centerY-hourHand.getHeight(), mBitmapPaint);
canvas.restore();

// draw minute hand
Bitmap minuteHand = mMinuteHandBitmapHolder.getBitmap();
canvas.save();
canvas.rotate(minRot, centerX, centerY);
canvas.drawBitmap(minuteHand, centerX-minuteHand.getWidth()/2f,
    centerY-minuteHand.getHeight(), mBitmapPaint);
canvas.restore();
...
```

8. Drawing the second hand is almost identical to the previous example.

In the wear module's `ConvergenceWatchFaceService.java`:

```
...
if (mAmbient) {
    // draw center of watch hands
    if(mBurnInProtection) {
        canvas.drawCircle(centerX, centerY, 6f, mBlackFillPaint);
    }
    canvas.drawCircle(centerX, centerY, 6f, mWhiteFillPaint);

    // draw background for peek card
    if(mPeekCardRect != null) {
        canvas.drawRect(mPeekCardRect, mBlackFillPaint);
    }
} else {
    // Only draw the second hand in interactive mode.
    mSecondHandLength = 100f / 320 * width;
    float secX = (float) Math.sin(secRotRad) * mSecondHandLength;
    float secY = (float) -Math.cos(secRotRad) * mSecondHandLength;
    canvas.drawLine(centerX, centerY, centerX + secX, centerY +
        secY, mSecondPaint);

    // draw center of watch hands
    canvas.drawCircle(centerX, centerY, 6f, mSecondPaint);
}
...
```

9. If continuous sweep is enabled, the watch face is visible, and ambient mode is enabled, invalidate the watch face at the end of `onDraw`. Invalidating `onDraw` every frame means that the watch face attempts to redraw as fast as possible. This behavior is what implements continuous sweep.

In the wear module's `ConvergenceWatchFaceService.java`:

```
...
if(mContinuousSweep && isVisible() && !isInAmbientMode()) {
    invalidate();
}
}
```

At this point, run the sample code and experiment with switching between ambient and interactive mode. I also recommend testing lowbit ambient and burn-in protection.

Summary

Implementing watch faces includes lot of boilerplate code, but once you understand how the samples in this chapter work, creating your own implementations will be straightforward. Android Wear allows for watch faces to display information that was previously inaccessible to watches. Watch faces are the first thing that users see when they look at their watches, so your designs have the potential of being seen more than traditional apps.

Part **IV**

Google Glass

Running Apps Directly on Glass

We'll start this chapter by comparing the two different ways in which we can develop apps for Glass: with the `Mirror API` and with the `Glass Development Kit (GDK)`. Although the rest of the book only demonstrates how to use the `GDK`, knowing about the `Mirror API` helps you understand the benefits and detriments of the `GDK`. We'll also discuss two patterns that are common in Glassware: immersions and ongoing tasks. Finally, we'll see how to implement immersions by developing a scrolling list of cards with a view called `WearableListView`.

The GDK and the Mirror API

The `GDK` and the `Mirror API` are two different development kits for creating Glassware. The `GDK` is an extension of the `Android SDK`, which was initially designed for developing handheld apps. In other words, your Glassware can leverage the entire `Android SDK`—from activities and services to the location and camera APIs—in addition to the functionality that the `GDK` provides exclusively for Glass such as:

- the ability to launch activities in response to voice commands,
- the ability to add cards to the timeline, and
- access to widgets and views designed specifically for Glass that allows you to create layouts with a style that is consistent with the rest of the platform.

Glassware based on the Mirror API, on the other hand, is not based on the Android SDK but rather on web applications that interact with Glass through the cloud. That is, this Glassware does not reside on Glass itself, but on a web server. For the most part, the Mirror API enables Glassware to

1. post timeline items,
2. receive notifications when users interact with a timeline item, and
3. intermittently receive coarse location updates.

To illustrate how users typically interact with Mirror API-based Glassware, consider CNN Breaking News, which periodically delivers videos of news articles as timeline items:

4. Users “install” CNN Breaking News by granting the web application permission to manage the timeline, perhaps on a website or through the MyGlass app.
5. CNN presumably schedules a news story to be pushed to all of the interested users.
6. When CNN pushes a story as a timeline item, its web application uses the Mirror API to send timeline items to all subscribed users.

When users receive a news story as a timeline item, they can tap on it to view the video.

Should I use the Mirror API or the GDK?

For the most part, the requirements of your application will determine whether you should build it with the GDK or the Mirror API. The advantages of the GDK and the Mirror API are listed below. They should help you decide which SDK to use.

Use the GDK when your Glassware:

- requires frequent GPS and orientation updates.
- needs real time interaction or custom layouts.
- must support offline usage.

Use the Mirror API when:

- periodically receiving coarse location is sufficient.
- basic interactions such as posting timeline items and responding to menu clicks is sufficient.
- online-only usage is sufficient.

Also note that you develop for the GDK in java while you can develop for the Mirror API in any web framework that supports RESTful APIs.

Additionally, Mirror API-based Glassware can be updated or modified without any user intervention since these applications reside on a web server. With GDK-based Glassware, on the other hand, Glass must download a new apk for every update.

In this book, we'll focus exclusively on GDK-based Glassware.

When users issue a voice command to start GDK-based Glassware, it can display content as part of the timeline (that is, as an ongoing task) or as an independent experience (that is, as an immersion). In the next section, we'll learn about both patterns.

Immersion and Ongoing Tasks

Immersion and ongoing tasks are two patterns that Glassware can follow. Let's start by covering both at a high level.

Immersion

Immersion are experiences that are independent of the timeline. That is, when an immersion starts, it does not appear as a timeline item but rather as a standalone activity. This activity can, in turn, start other activities just like an activity in a handheld app. Thus, an immersion is more than just a single activity: it is the experience provided by the collection of activities.

The Mini Games Glassware, which is available from the MyGlass app, is an example of an immersion.

To start Mini Games, use the "OK Glass... Play a game of" voice command and select "Clay shooter" (see Figure 8-1).

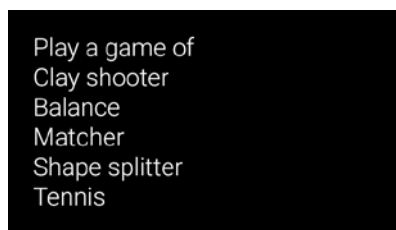


Figure 8-1. Starting the Mini Games Glassware

The voice command starts the immersion shown in Figure 8-2, which displays the game screen.

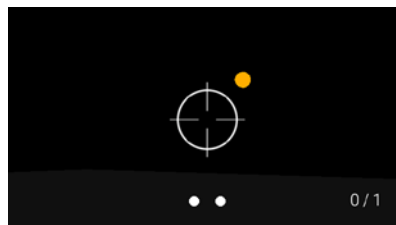


Figure 8-2. The Clay Shooter mini game

The game runs independently of the timeline and relies on head gestures to aim at clay targets. Users launch and shoot the clay targets by either tapping on the touchpad or saying, “pull” and “shoot”, respectively. Immersions should be important enough to warrant the user’s full attention, and a game certainly meets this criterion.

Furthermore, note that the timeline is inaccessible while using an immersion. That is, swiping to the left or to the right does not navigate between timeline items, but can have an effect on the immersion itself.

If the screen of Glass times out, the device automatically displays the home screen once a user wakes it up. Swiping down, which is a gesture analogous to pressing the back button in a handheld app, exits an activity. When users leave the final activity in the immersion, they return to the timeline.

Ongoing Tasks

Ongoing tasks are an alternative pattern in which the interface resides directly within the timeline. Ongoing tasks insert a timeline item called a LiveCard into the present and future section of the timeline (that is, to the left of the home screen). Users can scroll to and from a LiveCard by swiping left or right just as with any other timeline item. Ongoing tasks update the content of a LiveCard from a service. Additionally, a LiveCard can respond to user input such as tapping and performing head gestures, except for swiping left or right since these latter gestures are reserved for scrolling through timeline items. That is, unlike immersions, ongoing tasks are subject to the constraints of the timeline.

The compass app, which is available from the MyGlass app, is an example of an ongoing task.

To start the compass, use the “OK Glass... Show a compass” voice command, which inserts the LiveCard shown in Figure 8-3 into the present and future section of the timeline.

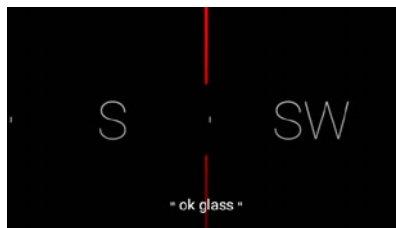


Figure 8-3. The compass Glassware creates a LiveCard that indicates the cardinal direction in which Glass is facing

The compass indicates the cardinal direction in which Glass is facing. To exit the compass, tap on the LiveCard to display the menu and choose stop.

As illustrated by the compass Glassware, ongoing tasks create a LiveCard that resides directly inside the timeline. Users can still scroll through the timeline and start other Glassware while an ongoing task is running. If the screen times out while displaying a LiveCard, the device immediately displays the LiveCard when it wakes up.

Combined Immersion and Ongoing Task

Applications can combine immersions and ongoing tasks. Although the initial voice action can only start an immersion or an ongoing task, immersions can start ongoing tasks and vice versa. The timer, which is available from the MyGlass app, is an example of an immersion that starts an ongoing task.

To start the timer, use the “OK Glass... Start a timer” voice command, which launches the immersion in Figure 8-4.

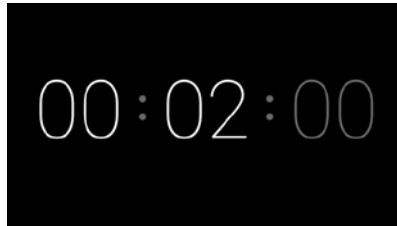


Figure 8-4. An immersion used to configure the timer’s ongoing task

Swipe to the left and to the right to increase or decrease the duration of the timer, which is presented in the format hours:minutes:seconds. Note that this screen is an immersion and not an ongoing task. Although voice commands can start ongoing tasks directly, the timer’s voice command starts an immersion that is used to configure the ongoing task.

Once you select the duration of the timer, tap on the immersion to open the options menu and select *Start* (see Figure 8-5).

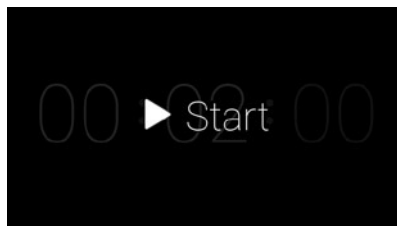


Figure 8-5. Starting the timer’s ongoing task

The system will then create a LiveCard that displays the time remaining before the timer goes off (see Figure 8-6). Although this screen looks similar to the previous immersion, it’s located within a timeline item and contains an options menu.



Figure 8-6. A LiveCard displays the time remaining before the timer goes off

To exit the timer, tap on the LiveCard to display the menu and choose “stop”.

Now that we understand the difference between `immersions` and `ongoing tasks`, let’s learn to implement them. We’ll cover `immersions` in the next section and `ongoing tasks` in the next chapter.

Getting Started

Before writing any code, you should know how to run and debug Glassware on Glass.

1. From the Android SDK Manager, install the latest version of the GDK.

On Glass, enable debug mode by tapping on the *Settings* card and then tapping on *Device Info* to access the menu. If needed, select the last menu item, which is labeled *Turn on debug*, as shown in Figure 8-7.

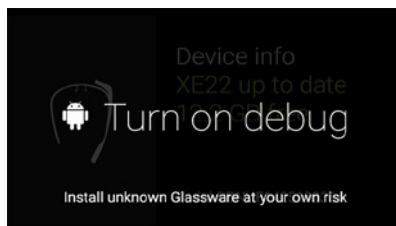


Figure 8-7. Turning on debug mode is necessary for developing with the GDK

2. Connect Glass to your computer and verify that `adb` devices can find the device. Setting up `adb` is outside the scope of this book, but if you have any issues I’ll shamelessly recommend my blog post on the topic:<http://www.ocdevelopers.com/2014/installing-native-android-apps-on-glass-adb-101/>.

Once you verify that adb can communicate with Glass, you're ready to code. Optionally, if you want to get rid of the cable and debug your Glassware over WiFi, you can set up adb over WiFi as follows.

1. Make sure both your computer and Glass are connected to the same network.
2. Temporarily connect your computer to Glass using a micro USB cable.
3. Find the IP Address of Glass with the following adb command:

```
adb shell ip addr show
```

On my setup, the command has the following output.

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ifb0: <BROADCAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN qlen 32
    link/ether 46:c2:1c:67:b3:fc brd ff:ff:ff:ff:ff:ff
    inet 192.168.167.239/0 brd 255.255.255.255 scope global ifb0
    inet6 fe80::44c2:1cff:fe67:b3fc/64 scope link
        valid_lft forever preferred_lft forever
3: ifb1: <BROADCAST,NOARP> mtu 1500 qdisc noop state DOWN qlen 32
    link/ether e6:f2:b5:78:b9:29 brd ff:ff:ff:ff:ff:ff
4: sit0: <NOARP> mtu 1480 qdisc noop state DOWN
    link/sit 0.0.0.0 brd 0.0.0.0
5: ip6tnl0: <NOARP> mtu 1452 qdisc noop state DOWN
    link/tunnel6 :: brd ::
6: p2p0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether 00:90:4c:33:22:11 brd ff:ff:ff:ff:ff:ff
7: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether f8:8f:ca:26:9e:e3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.8.108/24 brd 192.168.8.255 scope global wlan0
    inet6 fe80::fa8f:caff:fe26:9ee3/64 scope link
        valid_lft forever preferred_lft forever
```

While the output looks rather intimidating, we are only interested in the IP address given for wlan0, which is shown in bold. In my case, the IP Address of Glass is 192.168.8.108, but your value will likely be different.

4. Enable adb over TCP/IP on port 555 with the command:

```
adb tcpip 5555
```

5. With the IP Address obtained in step one, type the following command to enable adb over WiFi:

```
adb connect <ip address of Glass>
```

If you call `adb devices` with the USB cable still connected, you should be able to see two attached devices: Glass and Glass over WiFi.

```
List of devices attached
016837661101800B    device
192.168.8.108:5555  device
```

Then, disconnect the USB cable from Glass. Running `adb devices` once again should now only display Glass over WiFi as the only attached device. Even though the connection is running through WiFi, you can treat it just like a regular wired connection. Note that while debugging over WiFi is convenient, a downside is that your battery is not charging. Call `adb disconnect` to terminate the connection over WiFi.

Voice commands

Voice commands allow users to start Glassware from the home card by saying, for instance, “OK Glass... Start a timer.” Voice commands can either be listed or unlisted. Google has approved listed voice commands for use in Glassware and they can be used in production. Unlisted voice commands, in contrast, can only be used during development. Find the full list of listed voice commands at this URL <https://developers.google.com/glass/develop/gdk/reference/com/google/android/glass/app/VoiceTriggers.Command>. Note that Google can introduce new listed voice commands along with new versions of the operating system.

Listed commands ensure that voice commands are succinct, unique, and reasonable. If developers were allowed to use any voice command in production, the number of voice commands could become unwieldy and developers could unwittingly create redundant commands such as “Start a game” and “Play a game.” Moreover, Glass is able to optimize voice recognition for listed voice commands since they are finite in quantity. Even though listed voice commands are great for production, unlisted voice commands are useful for development and help you determine if your Glassware would benefit from new voice commands. If appropriate, you can suggest new voice commands to Google and hope that they agree your command is needed (see <https://developers.google.com/glass/distribute/voice-form>). Glassware that utilizes unlisted voice commands will not be approved for inclusion in the MyGlass store.

The following example demonstrates how to implement voice commands.

Implementing an Immersion

In this section, we’ll create an `immersion` that is started by an unlisted voice command.

Note The source code for this example is located in the `app` module of the `GlasswareBasics` project.

1. Run the module on Glass and start the “Hello World” voice command. In `AndroidManifest.xml`, delete the `android:theme` attribute from the application tag.

This step ensures that the Glass theme is applied instead of a handheld theme. Handheld themes should never be used on Glass because their fonts, colors, and appearance are not suited for the small screen of Glass. These themes may even display a tool bar (which was formerly known as an action bar) at the top of the screen.

2. Create `HelloWorldActivity` and give it a content view. For now, we’ll use a `TextView` with the text “Hello World”.

In `HelloWorldActivity.java`:

```
public class HelloWorldActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView helloWorld = new TextView(this);
        helloWorld.setText("Hello World");
        setContentView(helloWorld);
    }
}
```

Although our activity has content, we’re not currently able to launch it because it’s not associated with any voice commands. Next, we’ll see how to configure the Glassware to start the activity with a voice command.

3. Create a trigger, which specifies the keyword that a voice command uses. We’ll use “Hello World”.

In `res > xml > hello_trigger.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Hello World" />
```

4. In the manifest, request the `DEVELOPMENT` permission.

Unlisted voice commands require this permission or else they do not appear in the list of voice commands. This permission indicates that you are aware you cannot use unlisted voice commands in production. Glassware with unlisted voice commands cannot be approved for inclusion in the MyGlass store.

In `AndroidManifest.xml`:

```
<uses-permission
    android:name="com.google.android.glass.permission.DEVELOPMENT" />
```

5. Declare `HelloWorldActivity` to be launched in response to the “Hello World” voice command.

The `VOICE_TRIGGER` intent-filter action relies on a meta-data tag to find the trigger it should use. This meta-data tag is a child of the activity tag and references the trigger we created in step 3.

In `AndroidManifest.xml`:

```
<activity
    android:name=".HelloWorldActivity"
    android:icon="@drawable/ic_launcher">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>
    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/hello_trigger" />
</activity>
```

Now that the voice trigger is set, we can start the activity in one of two ways. First, use the “OK Glass... Hello World” voice command (see the top of Figure 8-8) or second, tap on the home card and scroll through the menu items until you find the *Hello World* item (see the bottom of Figure 8-8). Note that the menu item has an icon that represents the app. This icon was specified by the `android:icon` attribute of the activity declaration. Icons should be 50x50px and should only contain a white foreground on a transparent background.

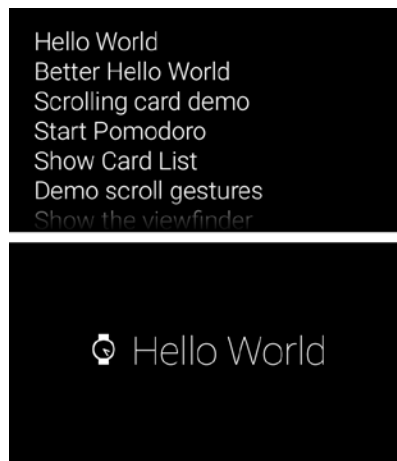


Figure 8-8. Launching the Glassware with the “Hello World” voice action. Using voice recognition (bottom). Using the voice command menu, which can be opened by tapping on the home screen (top)

When you start the Glassware, the immersion shown in Figure 8-9 will pop up.

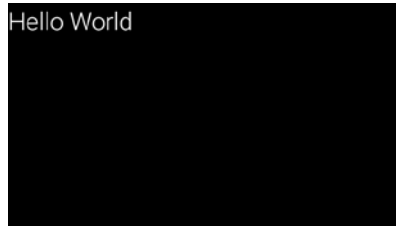


Figure 8-9. The activity launches in response to the “Hello world” voice command

Try tapping on the card or swiping to the left or to the right. Absolutely nothing will happen since we haven’t implemented any sort of interaction. However, even if an immersion doesn’t support any interaction, it should provide feedback to indicate that nothing will happen. The last example in this chapter demonstrates how to provide this feedback.

Using CardScrollView

CardScrollView is a UI widget included in the GDK that implements a list of horizontally scrolling cards similar to the timeline. For instance, the “Send message” voice command uses CardScrollView to display a list of a user’s contacts.

CardScrollView obtains its cards as views from a subclass of CardScrollAdapter, which must override the following methods.

- `int getCount()`: returns the number of items in the CardScrollView.
- `Object getItem(int position)`: returns an object that represents the item at the position (that is, index) given by the parameter.
- `View getView(int position, View convertView, ViewGroup parent)`: returns a View that displays the data at the position given by the first parameter. The second parameter, if not null, is an old view that should be reused, and the third parameter is the parent that the view will be attached to.
- `int getPosition(Object item)`: returns the index of the object given as the parameter. This method is the inverse of the `getItem` method.

CardScrollAdapter extends BaseAdapter, which is why these methods may look familiar. An example will demonstrate how to implement CardScrollAdapter shortly.

Playing System Sounds

Glassware should provide auditory feedback in response to user actions. The GDK contains several system sounds that should be used to provide consistent auditory feedback across Glassware. To play a system sound, call AudioManager’s `playSoundEffect` method:

```
AudioManager audioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
audioManager.playSoundEffect(effectType);
```

Where effectType is an int with one of the following values:

- **Sounds.DISALLOWED:** indicates that an action a user is taking is either not allowed or does not do anything.
- **Sounds.DISMISSED:** indicates that a user dismissed an item. This sound plays any time a user swipes down to dismiss a screen.
- **Sounds.ERROR:** indicates that an error occurred.
- **Sounds.SELECTED:** indicates that a user selected an item. For example, this sound plays when users swipe on the touchpad to select a new menu item.
- **Sounds.SUCCESS:** indicates that an action was completed successfully. For instance, this sound plays when a user successfully sends a message.
- **Sounds.TAP:** indicates that a user tapped on a card. For example, this sound plays when users tap on the home screen.

The next example demonstrates how to play the TAP sound when users tap on a CardScrollView.

Implementing Scrolling Cards

This example displays a list of strings in a CardScrollView, where each string corresponds to a single card.

1. In AndroidManifest.xml, delete the android:theme attribute from the application tag and request the DEVELOPMENT permission.

In AndroidManifest.xml:

```
<uses-permission
    android:name="com.google.android.glass.permission.DEVELOPMENT" />
```

2. Create a trigger, which specifies the keyword that a voice command uses. We'll use "Scrolling card demo."

In res > xml > scroll_demo_trigger.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Scrolling card demo" />
```

3. Declare ScrollingCardsActivity to be launched in response to the "Scrolling card demo" voice command.

In `AndroidManifest.xml`:

```
<activity
    android:name=".ScrollingCardsActivity"
    android:icon="@drawable/ic_launcher">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>
    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/scroll_demo_trigger" />
</activity>
```

4. Next, create a subclass of `CardScrollAdapter` that creates child views for every item in the list.

This class, which is an inner class of `ScrollingCardsActivity`, takes a list of strings as a parameter and creates a single child view per string.

In `ScrollingCardsActivity.java`:

```
private static class TextCardAdapter extends CardScrollAdapter {
    private Context mContext;
    private List<String> mItems;

    public TextCardAdapter(Context context, List<String> items) {
        mContext = context;
        mItems = items;
    }

    @Override
    public int getCount() {
        return mItems.size();
    }

    @Override
    public Object getItem(int position) {
        return mItems.get(position);
    }

    @Override
    public View getView(int position, View view, ViewGroup parent) {
        TextView textView;

        if(view == null) {
            textView = new TextView(mContext);
        } else {
            textView = (TextView)view;
        }
    }
}
```

```

        textView.setText(mItems.get(position));

        return textView;
    }

    @Override
    public int getPosition(Object item) {
        return mItems.indexOf(item);
    }
}

```

TextCardAdapter should be self explanatory because of its similarity with a standard list adapter.

5. Declare and initialize member variables.

The onCreate method generates a list of strings and using them to create TextCardAdapter. Note that CardScrollView lets you set an item click listener just like a ListView. We'll implement this listener in the next step.

In ScrollingCardsActivity.java:

```

public class ScrollingCardsActivity extends Activity {
    private CardScrollView mCardScrollView;
    private List<String> mWords;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mWords = new ArrayList<String>();
        mWords.add("one");
        mWords.add("two");
        mWords.add("three");
        mWords.add("four");
        mWords.add("five");
        mWords.add("six");
        mWords.add("seven");

        TextCardAdapter textCardAdapter = new TextCardAdapter(this, mWords);
        mCardScrollView = new CardScrollView(this);
        mCardScrollView.setAdapter(textCardAdapter);
        mCardScrollView.setOnItemClickListener(itemClickListener);

        setContentView(mCardScrollView);
    }
    ...
}

```

6. Play the TAP sound and display a toast message when a user taps on CardScrollView

In `ScrollingCardsActivity.java`:

```
private AdapterView.OnItemClickListener itemClickListener =
    new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
            AudioManager audioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
            audioManager.playSoundEffect(Sounds.TAP);

            Toast.makeText(ScrollingCardsActivity.this, "Clicked on item " + mWords.get(position),
                Toast.LENGTH_SHORT).show();
        }
    };
```

7. Activate and deactivate the `CardScrollView` on `onResume` and `onPause`, respectively.

`CardScrollView` only responds to user input while activated. There is no need to process user input while paused.

In `ScrollingCardsActivity.java`:

```
@Override
protected void onResume() {
    super.onResume();
    mCardScrollView.activate();
}

@Override
protected void onPause() {
    mCardScrollView.deactivate();
    super.onPause();
}
```

Run the app with the “scrolling card demo” voice command and you will see a list of scrolling cards, the first two of which are shown in [Figure 8-10](#).

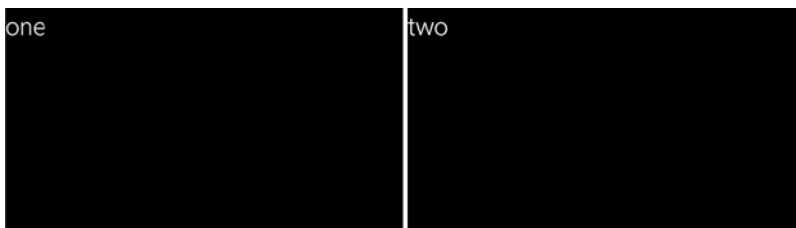


Figure 8-10. The `CardScrollView` from the example displays a list of card, each of which contains a string

Now that we understand how to use `CardScrollView`, we can make the “Hello world” example provide feedback when users try to tap or swipe on the touchpad.

Providing Feedback

Earlier, we mentioned that the “Hello world” sample should provide feedback that indicates it doesn’t respond to tapping or swiping. Providing feedback for a lack of action is just as important as providing feedback for an action. Doing so prevents your Glassware from appearing unresponsive when users try to interact with it. In particular, we want to provide the following feedback:

1. Play the DISALLOWED audio when a user taps on the card.
2. Make the screen tuggable in response to swiping. That is, the card should begin to move in the direction the user swipes, but only enough to indicate the card will not move any further, and then bounce back to its initial position. This effect is also called “rubber banding” or “inertial scrolling.”

This example, which we’ll call “Better Hello World,” is based on the “Hello world” example and incorporates these two types of feedback. We’ll implement tuggable feedback with a `CardScrollView` that only has a single item. `CardScrollView` takes care of providing the animation and doing all the hard work. The official GDK documentation even provides a class called `TuggableView` that wraps a single item `CardScrollView` (from <https://developers.google.com/glass/develop/gdk/card-scroller>).

3. Implement `TuggableView`.

This class extends `CardScrollView` and takes the resource ID of a layout as a parameter. It then inflates this layout and places it inside the `CardScrollView`'s single item. This approach is convenient since `CardScrollView` already implements inertial feedback.

In `TuggableView.java`:

```
public class TuggableView extends CardScrollView {

    private final View mContentView;

    /**
     * Initializes a TuggableView that uses the specified layout
     * resource for its user interface.
     */
    public TuggableView(Context context, int layoutResId) {
        this(context, LayoutInflater.from(context)
            .inflate(layoutResId, null));
    }

    /**
     * Initializes a TuggableView that uses the specified view
     * for its user interface.
     */
}
```

```

public TuggableView(Context context, View view) {
    super(context);

    mContentView = view;
    setAdapter(new SingleCardAdapter());
    activate();
}

/**
 * Overridden to return false so that all motion events still
 * bubble up to the activity's onGenericMotionEvent() method after
 * they are handled by the card scroller. This allows the activity
 * to handle TAP gestures using a GestureDetector instead of the
 * card scroller's OnItemClickListener.
 */
@Override
protected boolean dispatchGenericFocusedEvent(MotionEvent event) {
    super.dispatchGenericFocusedEvent(event);
    return false;
}

/** Holds the single "card" inside the card scroll view. */
private class SingleCardAdapter extends CardScrollAdapter {

    @Override
    public int getPosition(Object item) {
        return 0;
    }

    @Override
    public int getCount() {
        return 1;
    }

    @Override
    public Object getItem(int position) {
        return mContentView;
    }

    @Override
    public View getView(int position, View recycleView,
        ViewGroup parent) {
        return mContentView;
    }
}
}

```

4. In `AndroidManifest.xml`, delete the `android:theme` attribute from the application tag and request the `DEVELOPMENT` permission.

In `AndroidManifest.xml`:

```
<uses-permission
    android:name="com.google.android.glass.permission.DEVELOPMENT" />
```

5. Create a trigger, which specifies the keyword that a voice command uses. We'll use the "Better Hello World" keyword.

In `res > xml > better_hello_trigger.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Better Hello World" />
```

6. Start `BetterHelloWorldActivity` in response to the "Better Hello World" voice command.

In `AndroidManifest.xml`:

```
<activity
    android:name=".BetterHelloWorldActivity"
    android:icon="@drawable/ic_launcher">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>
    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/better_hello_trigger" />
</activity>
```

7. In the `onCreate` method, place the desired content inside a `TuggableView`. Then, set the `TuggableView` as the activity's content view.

In `BetterHelloWorldActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    TextView helloWorld = new TextView(this);
    helloWorld.setText("Hello World");

    TuggableView tuggableView = new TuggableView(this, helloWorld);
    tuggableView.setOnItemClickListener(mItemClickListener);

    setContentView(tuggableView);
}
```

8. When a user taps on the touchpad, play the `DISALLOWED` sound.

In `BetterHelloWorldActivity.java`:

```
private AdapterView.OnItemClickListener mItemClickListener =
    new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        AudioManager audioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
        audioManager.playSoundEffect(Sounds.DISALLOWED);
    }
};
```

Providing this additional feedback clearly indicates that this example does not respond to tapping or swiping. Any time you develop an immersion that does not respond to tapping on the touchpad, it should play the TAP sound. Similarly, all immersions that don't respond to swiping should use a `TappableView`.

Summary

We started this chapter by comparing both types of development for Glass: using the `Mirror` API and using the `GDK`. Although this book focuses exclusively on the `GDK`, understanding the `Mirror` API is important so you can decide which SDK is best for your Glassware. Then, we discussed immersions and ongoing tasks, which are the two main patterns for Glassware interfaces, and we implemented three examples that show how to create immersions and launch them with voice commands. We also learned to use a widget called `CardScrollView`, which displays a list of horizontally scrolling cards. The cards we developed in this chapter look rather ugly because they only consist of a `TextView` with its default settings. The default settings of a `TextView`, including font size and font family, are optimized for Android handhelds and don't work well with Glass. In the next chapter, we'll learn to create cards with the style used by the rest of the platform. We'll also see how to implement ongoing tasks with `LiveCards`.

Glass User Interface Essentials

In the previous chapter, we created Glassware with the GDK and learned to implement *immersions*, which are experiences that run independently of the timeline as a collection of one or more activities. However, our examples had unsightly user interfaces because they used `TextViews` with a default appearance that is intended for use with handheld devices. In this chapter, we'll start by learning to use the `CardBuilder` class to create interfaces with an appearance that matches the rest of the Glass platform. Then, we'll cover another common pattern for Glassware, *ongoing tasks*, in which content that updates in real-time is inserted directly into the timeline. In contrast to *immersions*, users can still navigate through timeline items while using *ongoing tasks*. Finally, we'll learn about a class called `Slider` that lets Glassware display progress and status.

Building Cards Styled for Glass

If a Glassware's user interface looks like the rest of the Glass platform, then users can easily infer what actions they can perform and what outcomes they can expect. That is, users expect interfaces with a similar appearance to have similar behavior. For instance, a menu is easily recognizable by its appearance, which consists of vertically centered text on a translucent overlay. When users recognize a menu, they know that there are three possible actions:

1. tap on the menu item to select it,
2. swipe to the left or right to select another menu item, or
3. swipe down to dismiss the menu.

The Glass platform uses specific fonts, grids, and colors to style cards (for details, see <https://developers.google.com/glass/design/style>). However, the GDK allows us to implement cards with this style without understanding the details of their design. The GDK provides a class called `CardBuilder` that takes care of properly styling cards in a variety of formats.

In this section, we'll create a scrolling list of cards to demonstrate the use of `CardBuilder` and to observe the variety of styles it supports.

Note The source code for this example is located in the app module of the `GlassUiEssentials` project. Run the module on the handheld device and start the “Show Card List” voice command.

Declaring a Voice Command

Start by creating a voice command that starts `CardListActivity`.

1. Create a voice trigger with a keyword of “Show Card List.”

In `res > xml > card_list_trigger.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Show Card List" />
```

2. Declare `CardListActivity` to be launched in response to the “Show Card List” voice command.

In `AndroidManifest.xml`:

```
<activity
    android:name=".CardListActivity"
    android:icon="@drawable/ic_launcher">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>
    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/card_list_trigger" />
</activity>
```

Extending CardScrollAdapter

In `CardListActivity`, a `CardScrollView` displays a list of scrolling cards that demonstrate several different card styles. Every card in this list is created with the `CardBuilder` class. The `CardListScrollAdapter` class, which extends `CardScrollAdapter`, receives a list of `CardBuilder` objects in its constructor and uses them to create a variety of cards that are displayed in `CardScrollView`.

1. Store the list of `CardBuilder` objects in a member variable.

In `CardListScrollAdapter.java`:

```
public static class CardListScrollAdapter extends CardScrollAdapter {
    private List<CardBuilder> mCards;

    public CardListScrollAdapter(List<CardBuilder> cards) {
        mCards = cards;
    }
    ...
}
```

2. Override `getCount` and have it return the number of items in the list.

In `CardListScrollAdapter.java`:

```
@Override
public int getCount() {
    return mCards.size();
}
```

3. Override the `getItem` method and have it return the `CardBuilder` object at the given position.

In `CardListScrollAdapter.java`:

```
@Override
public Object getItem(int position) {
    return mCards.get(position);
}
```

4. Override the `getViewTypeCount` method and have it delegate to `CardBuilder`.

In the previous chapter, we created a `CardScrollView` that only used a single type of view. In other words, every item in the list had the exact same layout and only differed in content. The views generated by `CardBuilder` in this example, on the other hand, can have multiple different layouts. The `CardScrollView` class needs to know how many different types of views are used in the list. This information allows `CardScrollView` to optimize rendering by recycling views.

The `getViewTypeCount` method returns the number of different views in a list. The return value of this method defaults to one, which is why we didn't need to override it in the example of the previous chapter. In the current example, the number of different views generated by `CardBuilder` is returned by the `CardBuilder.getViewTypeCount` method.

In `CardListScrollAdapter.java`:

```
@Override
public int getViewTypeCount() {
    return CardBuilder.getViewTypeCount();
}
```

5. Override the `getItemViewType` method to return an integer that uniquely identifies the view type used at a particular position.

This information allows `CardScrollView` to optimize rendering by recycling views. By default, this method returns 0 and does not need to be overridden if every item in the list is the same. In this case, we must override this method because `CardBuilder` can generate multiple types of layouts. We delegate to `CardBuilder`'s `getItemViewType` method, which uniquely identifies the type of view that the instance of `CardBuilder` generates.

In `CardListScrollAdapter.java`:

```
@Override
public int getItemViewType(int position) {
    return mCards.get(position).getItemViewType();
}
```

6. Override `getView` to return the views generated by `CardBuilder`.

This method delegates the creation of the view to the appropriate `CardBuilder` instance. Note that the `CardBuilder`'s `getView` method accepts `convertView` as a parameter, so it handles recycling views.

In `CardListScrollAdapter.java`:

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    return mCards.get(position).getView(convertView, parent);
}
```

7. Override the `getPosition` method to return the position of a given item within the list. In this case, an item is an instance of `CardBuilder`.

Note that this method is the inverse of the `getItem` method from step 3.

In `CardListScrollAdapter.java`:

```
@Override
public int getPosition(Object item) {
    return mCards.indexOf(item);
}
```

The methods from this class allow `CardScrollView` to obtain the views that are used to create the items of the scrolling list of cards.

Creating the List of Scrolling Cards

Now that we've extended `CardScrollView`, let's write the main content of `CardListActivity`. This activity displays a scrolling list of cards, where each card illustrates a layout that can be generated with `CardBuilder`.

1. Declare and initialize member variables.

When the `onCreate` method instantiates `CardListScrollAdapter`, it uses a list of `CardBuilder` objects that is generated in the `initCards` method. The `initCards` method demonstrates the use of `CardBuilder` and is implemented throughout the rest of this example.

In `CardListActivity.java`:

```
public class CardListActivity extends Activity {
    private CardScrollView mCardScrollView;
    private List<CardBuilder> mCards;
    private AudioManager mAudioManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mAudioManager = (AudioManager) getSystemService(AUDIO_SERVICE);

        initCards();
        mCardScrollView = new CardScrollView(this);
        CardListScrollAdapter adapter = new CardListScrollAdapter(mCards);
        mCardScrollView.setAdapter(adapter);
        mCardScrollView.setOnItemClickListener(mItemClickListener);

        setContentView(mCardScrollView);
    }
    ...
}
```

2. Play the `DISALLOWED` sound when users tap on a card to indicate that tapping does not do anything.

In `CardListActivity.java`:

```
private AdapterView.OnItemClickListener mItemClickListener =
    new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
            mAudioManager.playSoundEffect(Sounds.DISALLOWED);
        }
    };
```

3. Activate and deactivate `CardScrollView` in `onResume` and `onPause`, respectively.

This step makes sure that `CardScrollView` is activated when the activity is in the foreground and deactivated otherwise.

In `CardListActivity.java`:

```
@Override
protected void onResume() {
    super.onResume();
    mCardScrollView.activate();
}
```

```
@Override
protected void onPause() {
    mCardScrollView.deactivate();
    super.onPause();
}
```

4. Create the `initCards` method.

For now, this method is a placeholder for the code that instantiates `CardBuilder` objects with different types of layouts.

In `CardListActivity.java`:

```
private void initCards() {
    mCards = new ArrayList<CardBuilder>();
    ...
}
```

Below, we'll go through the different kinds of layouts that are currently available with `CardBuilder`. These layouts use image resources that are available with the sample project.

The TEXT Layout

The `TEXT` layout places a string of text at the top of a card, a footnote at the bottom left, and a timestamp at the bottom right as shown in Figure 9-1.

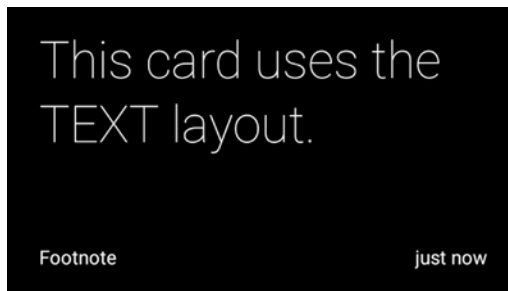


Figure 9-1. A card with the `TEXT` layout

The text dynamically resizes to fit the available space. In particular, the `TEXT` layout chooses the largest among several predefined text sizes so that the entire text fits in the view. If the text is too large to fit on the screen even with a small font size, `CardBuilder` cuts off the text near the end and adds ellipsis to indicate the text is incomplete. This process is called ellipsizing text.

To add this card to the list in the previous section, use the following code.

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText("This card uses the TEXT layout.")
    .setFootnote("Footnote")
    .setTimestamp("just now"));
```

Cards with the TEXT layout can optionally include an image as a full screen background (see Figure 9-2).

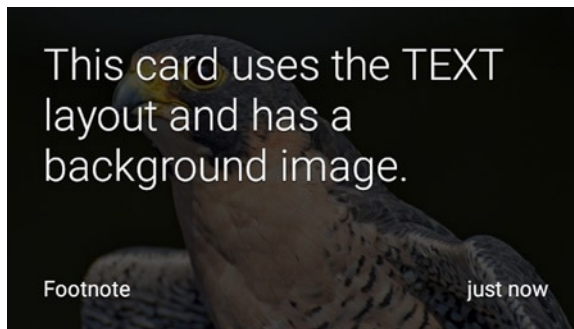


Figure 9-2. A card using the TEXT layout with an image in the background. The layout automatically applies a translucent overlay over the image to increase the contrast of the text

To include a background image, add a bitmap, drawable, or drawable resource ID to the `CardBuilder` with the `addImage` method:

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText("This card uses the TEXT layout and has a background image.")
    .setFootnote("Footnote")
    .setTimestamp("just now")
    .addImage(R.drawable.falcon));
```

Where the drawable given to `addImage` is of size 640x360 pixels.

The TEXT layout automatically covers the original background image with a translucent overlay that darkens the entire image and ensures that the white text has sufficient contrast. Note that the original image (see Figure 9-3) is significantly brighter than in the layout's background.



Figure 9-3. The original background image used in the card layout

The TEXT layout can also use a mosaic of multiple images as its background. The card in Figure 9-4, for instance, contains a mosaic of four images.

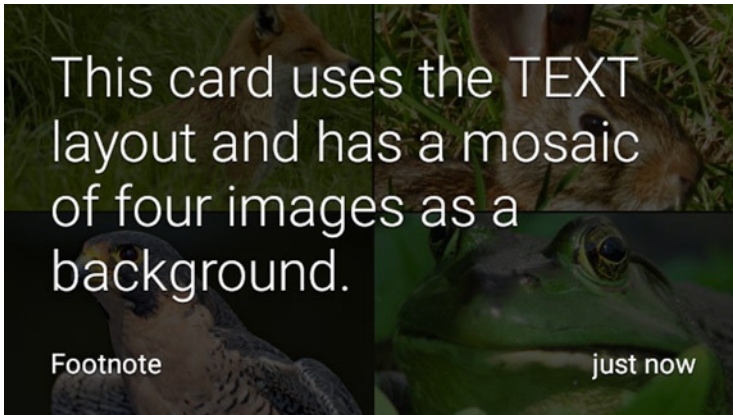


Figure 9-4. A card using the TEXT layout contains a mosaic of four images as a background

To add a mosaic of images, add multiple images to the `CardBuilder` using the `addImage` method:

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText("This card uses the TEXT layout and has a mosaic of four images as a
background.")
    .setFootnote("Footnote")
    .setTimestamp("just now")
    .addImage(R.drawable.fox)
    .addImage(R.drawable.hare)
    .addImage(R.drawable.falcon)
    .addImage(R.drawable.toad));
```

A background mosaic can contain up to eight images, as shown in Figure 9-5.

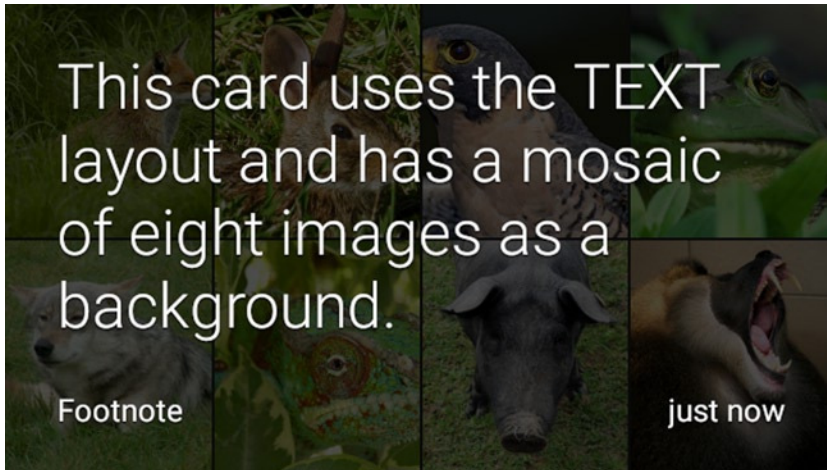


Figure 9-5. A card using the TEXT layout with a mosaic of eight images as a background

This code demonstrates how to create a TEXT layout with a mosaic of eight images as its background:

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText("This card uses the TEXT layout and has a mosaic of eight images as a
background.")
    .setFootnote("Footnote")
    .setTimestamp("just now")
    .addImage(R.drawable.fox)
    .addImage(R.drawable.hare)
    .addImage(R.drawable.falcon)
    .addImage(R.drawable.toad)
    .addImage(R.drawable.wolf)
    .addImage(R.drawable.chameleon)
    .addImage(R.drawable.pig)
    .addImage(R.drawable.monkey));
```

The TEXT_FIXED Layout

The TEXT_FIXED layout is identical to the TEXT layout except that its text size is fixed at 30px, which is a relatively small value. This layout should be used when displaying a long string of text across multiple cards to ensure that the text size between cards is consistent. Figure 9-6 shows an example of a TEXT_FIXED layout.

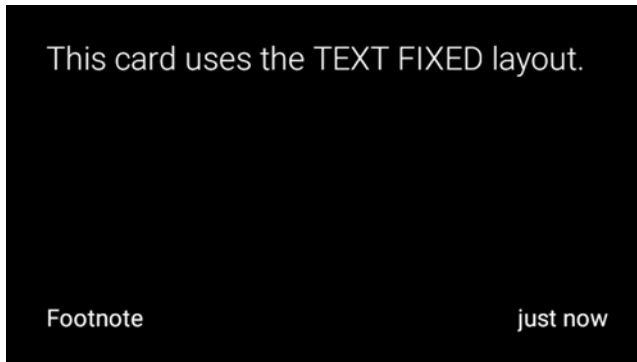


Figure 9-6. A card using the `TEXT_FIXED` layout

This snippet shows how to create a card with the `TEXT_FIXED` layout.

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.TEXT_FIXED)
    .setText("This card uses the TEXT FIXED layout. The text size will automatically
    adjust to best fit the available space.")
    .setFootnote("Footnote")
    .setTimestamp("just now"));
```

Although the `TEXT` and `TEXT_FIXED` layouts can display background images, these backgrounds are intentionally muted and not prominent. If you want to display an image with more emphasis, one of the next layouts is a better choice.

The `COLUMNS` Layout

This layout divides the screen into two sections: the left section contains one or more images while the right section contains text, a footnote, and a timestamp. Unlike the background images of the `TEXT` and `TEXT_FIXED` layouts, the images here do not have a translucent overlay. The card in Figure 9-7 uses a `COLUMNS` layout with a single image.

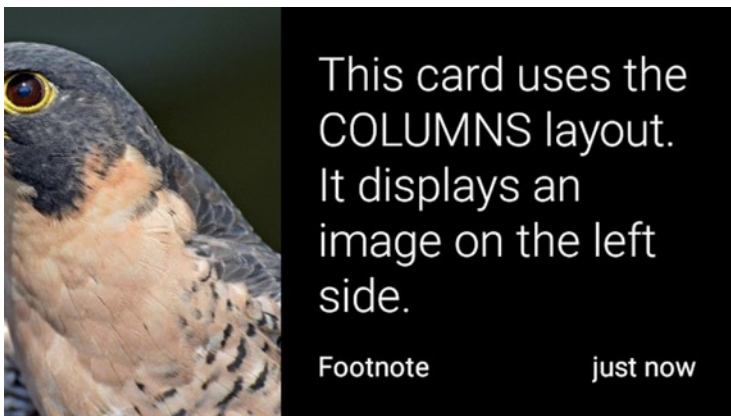


Figure 9-7. A card using the `COLUMNS` layout with an image

This layout contains a picture of an eagle, but its face is cut off. The COLUMNS layout center-crops all of its images to ensure that they fill their entire space while maintaining its aspect ratio, and the parts of an image that fall outside its bounds are clipped. The only way to center the eagle on the layout would be to give `CardBuilder` a different image in which the position of the eagle's head is closer to the center of the image.

This snippet shows how to create a card with the COLUMNS layout.

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.COLUMNS)
    .setText("This card uses the COLUMNS layout. It displays an image on
the left side.")
    .setFootnote("Footnote")
    .setTimestamp("just now")
    .addImage(R.drawable.falcon));
```

The COLUMNS layout can also use a mosaic of images, as shown in Figure 9-8.

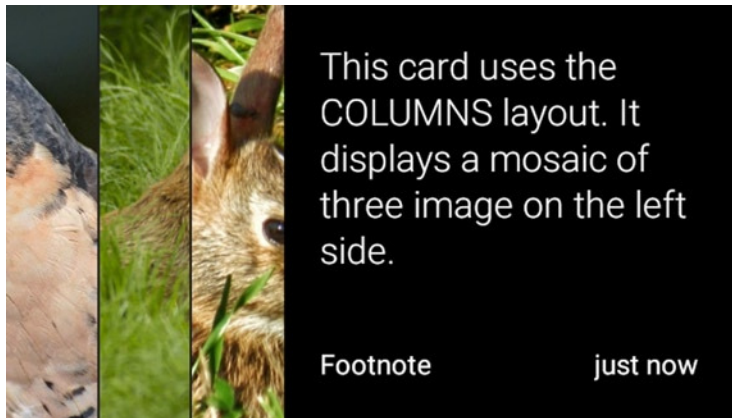


Figure 9-8. A card using the COLUMNS layout with a mosaic of three images

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.COLUMNS)
    .setText("This card uses the COLUMNS layout. It displays a mosaic of three
image on the left side.")
    .setFootnote("Footnote")
    .setTimestamp("just now")
    .addImage(R.drawable.falcon)
    .addImage(R.drawable.fox)
    .addImage(R.drawable.hare));
```

In addition to containing images, the side of a COLUMNS layout can contain an icon, as shown in Figure 9-9.

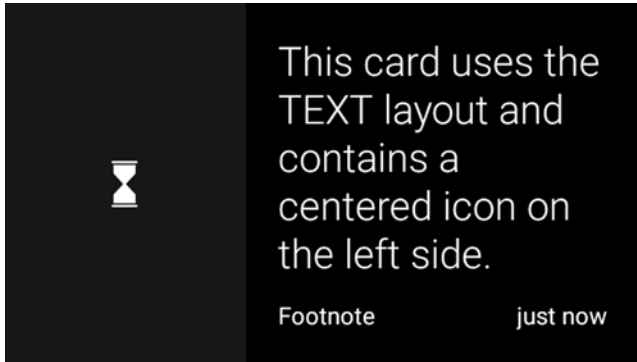


Figure 9-9. A card using the `COLUMNS` layout with an icon

To add an icon to the `COLUMNS` layout, call the `setIcon` method and pass it a bitmap, a drawable, or a drawable resource ID.

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.COLUMNS)
    .setText("This card uses the TEXT layout and contains a centered icon on
    the left side.")
    .setFootnote("Footnote")
    .setTimestamp("just now")
    .setIcon(R.drawable.ic_timer_50));
```

The `COLUMNS_FIXED` Layout

The `COLUMNS_FIXED` layout is almost identical to the `COLUMNS` layout except that it displays the text at a fixed size of 40 pixels. Use this layout when displaying multiple adjacent cards of this type to keep the size of the text consistent. Figure 9-10 contains an example of this layout.

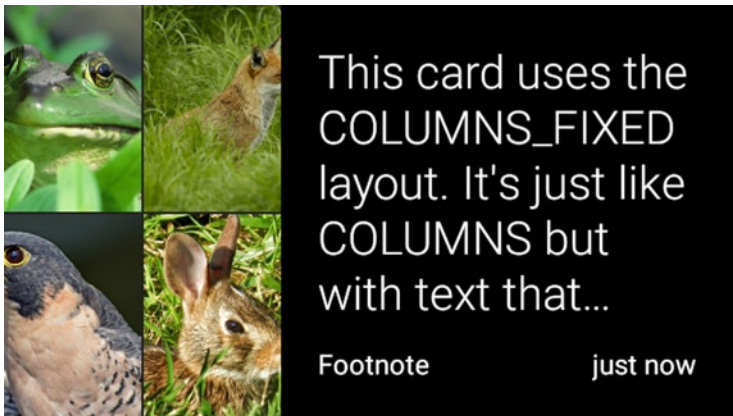


Figure 9-10. A card using the `COLUMNS_FIXED` layout with a mosaic of four images

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.COLUMNS_FIXED)
    .setText("This card uses the COLUMNS_FIXED layout. It's just like COLUMNS but with
        text that automatically resizes to fit the available space.")
    .setFootnote("Footnote")
    .setTimestamp("just now")
    .addImage(R.drawable.toad)
    .addImage(R.drawable.fox)
    .addImage(R.drawable.falcon)
    .addImage(R.drawable.hare));
```

Although the `COLUMNS` layout displays an image in the foreground, the image only occupies the leftmost column. If the image is the primary focus of a card, use the `CAPTION` layout instead.

The CAPTION Layout

The `CAPTION` layout displays a full screen image in the foreground along with a short caption, a footnote, and a timestamp (see Figure 9-11).

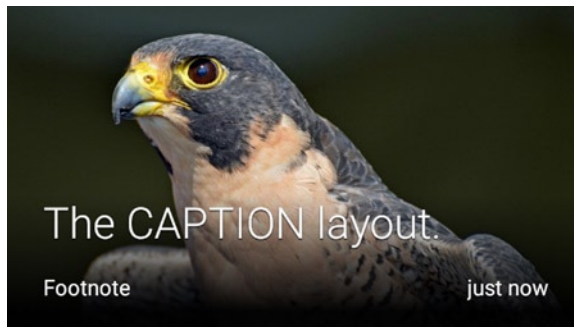


Figure 9-11. A card using the `CAPTION` layout

This layout displays a gradient overlay on top of the image to ensure that the footnote, timestamp, and caption text have sufficient contrast. A gradient overlay gradually transitions from opaque at the bottom of the layout to transparent at the top of the layout. This overlay gives the text a relatively dark background without obscuring the rest of the image.

This snippet creates a card with a `CAPTION` layout.

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.CAPTION)
    .setText("The CAPTION layout.")
    .setFootnote("Footnote")
    .setTimestamp("just now")
    .addImage(R.drawable.falcon));
```

The TITLE layout

The TITLE layout displays a full screen image, a title, and an optional icon as shown in Figure 9-12.



Figure 9-12. A card using the TITLE layout

The “Send message” voice command, for example, uses the TITLE layout to display a list of contacts. In this case, the image shows a contact’s avatar, the title contains a contact’s name, and a star icon is placed next to contacts who are in the “favorites” category.

Also note that the TITLE layout places a gradient overlay on top of the image, just like the CAPTION layout.

This snippet creates a card with a TITLE layout that uses an icon.

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.TITLE)
    .setText("The TITLE layout")
    .setIcon(R.drawable.ic_timer_50)
    .addImage(R.drawable.falcon));
```

The AUTHOR Layout

The AUTHOR layout displays a message along with an avatar, heading, and subheading to describe the author, as shown in Figure 9-13.



Figure 9-13. A card using the AUTHOR layout

In this case, the author's name and location are included as the card's heading and subheading, respectively. The message is specified with `CardBuilder`'s `setText` method. Also note that the AUTHOR layout automatically clips the author's avatar circular shape.

This snippet demonstrates how to create a card with an AUTHOR layout.

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.AUTHOR)
    .setText("The AUTHOR layout displays an author's avatar, name, and subheading along
    with a message. A twitter message could use this layout.")
    .setIcon(R.drawable.fox)
    .setHeading("Fox Vulpes")
    .setSubheading("San Diego, California")
    .setFootnote("Footnote")
    .setTimestamp("just now"));
```

Use the AUTHOR layout when displaying a single long message posted by a single user. For instance, a short chat message may be better suited for a COLUMNS layout while a twitter message would look better with the AUTHOR layout.

The MENU Layout

The MENU layout is the same one used in menus throughout the Glass platform (see Figure 9-14). These layouts display an icon, the menu item's title, and an optional description of the menu item that appears in smaller text near the bottom of the card.



Figure 9-14. A card using the MENU layout

For instance, every card in the list of voice commands (which you can open by tapping on the home screen) uses the MENU layout without specifying a description. Another example is the menu used in the “Auto Backup” card within the settings of Glass, which uses a MENU layout with a description to specify how many items will be backed up (such as “7 items will be backed up”).

This snippet demonstrates how to create a card with a MENU layout that uses a description. Note that the description is specified with the `setFootnote` method.

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.MENU)
    .setText("The MENU layout")
    .setIcon(R.drawable.ic_timer_50)
    .setFootnote("menu item description here"));
```

The ALERT Layout

The ALERT layout displays a status or error message of significant importance (see Figure 9-15). For instance, the “Send message” voice command uses an ALERT layout to indicate when voice recognition can’t detect any words.

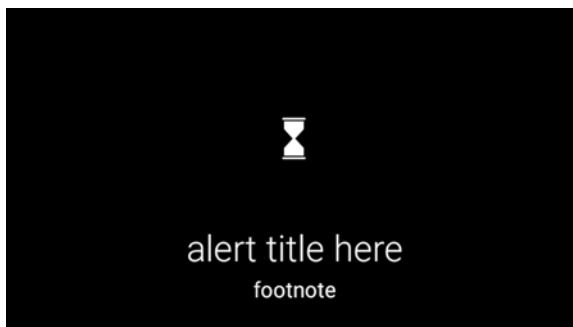


Figure 9-15. A dialog with an ALERT layout used by the “Send message” voice command

This “Send message” dialog prompts users to tap on the card to speak again. Alternatively, users can swipe down to dismiss the dialog.

This section shows how to create a dialog with an ALERT layout when a user taps on a card. Additionally, the example demonstrates how to detect when users tap on this card.

Implementing GlassAlertDialog

`GlassAlertDialog`, which extends the `Dialog` class, is an inner class of `CardListActivity`. In Glass, all dialogs occupy the entire screen and do not contain any buttons. `GlassAlertDialog` displays a message using the ALERT layout, and users can dismiss the dialog by swiping down. Additionally, this dialog displays a message when users tap on the dialog.

1. Create the `DialogTheme` style, which places a translucent background behind the dialog to ensure the ALERT layout’s text has sufficient contrast with the background while still revealing the content underneath the dialog.

In `res > values > colors.xml`:

```
<color name="dialog_bgcolor">#B2000000</color>
```

In `res > values > styles.xml`:

```
<style name="DialogTheme" parent="@android:style/Theme.DeviceDefault.Dialog">
  <item name="android:windowBackground">@color/dialog_bgcolor</item>
</style>
```

2. Declare member variables.

In `CardListActivity.java`:

```
public static class GlassAlertDialog extends Dialog {
    private OnClickListener mClickListener;
    private AudioManager mAudioManager;
    private GestureDetector mGestureDetector;
    ...
}
```

3. Initialize member variables.

`AudioManager` is used to play the TAP sound when users tap on the card and `GestureDetector` figures out when users taps on the touchpad. `GestureDetector` is covered in more detail in chapter 10.

This constructor also sets the content view of the dialog to an ALERT layout.

In `CardListActivity.java`:

```
public GlassAlertDialog(Context context, int iconRes, String text, String footnote) {
    super(context, R.style.DialogTheme);

    mAudioManager = (AudioManager) context.getSystemService(Context.AUDIO_SERVICE);
    mGestureDetector = new GestureDetector(context);
    mGestureDetector.setBaseListener(mBaseListener);
}
```

```

    setContentView(new CardBuilder(context, CardBuilder.Layout.ALERT)
        .setIcon(iconRes)
        .setText(text)
        .setFootnote(footnote)
        .getView());
}

```

4. Pass generic motion events to GestureDetector.

Generic motion events are generated every time a user interacts with the touchpad and are very similar to the touch events generated by handheld devices. GestureDetector must receive these generic motion events to figure out when a user taps on the touchpad.

In CardListActivity.java:

```

@Override
public boolean onGenericMotionEvent(MotionEvent event) {
    return super.onGenericMotionEvent(event) || mGestureDetector.onMotionEvent(event);
}

```

5. Implement GestureDetector.BaseListener.

When users tap on the dialog, play the TAP sound, trigger GlassAlertDialog's click listener, and dismiss the dialog. Note that dialogs in Glass automatically play the Sounds.DISMISSED sound when dismissed (or Sounds.DISALLOWED if not cancelable), which is why we only need to worry about playing the TAP sound.

In CardListActivity.java:

```

private GestureDetector.BaseListener mBaseListener = new GestureDetector.BaseListener() {
    @Override
    public boolean onGesture(Gesture gesture) {
        if(gesture == Gesture.TAP) {
            mAudioManager.playSoundEffect(Sounds.TAP);

            if(mClickListener != null) {
                // Glass dialogs don't have buttons. thus, which is always 0
                mClickListener.onClick(GlassAlertDialog.this, 0);
            }

            dismiss();
            return true;
        }

        return false;
    }
};

```

6. Create a getter and a setter for the onClickListener.

In `CardListActivity.java`:

```
public OnClickListener getOnClickListener() {
    return mClickListener;
}

public void setOnClickListener(OnClickListener onClickListener) {
    mClickListener = onClickListener;
}
```

Now that we've implemented a dialog, we must start it from `CardListActivity`.

Displaying GlassAlertDialog

We'll add a card to the end of the scrolling list of cards from `CardListActivity` that prompts users to tap on the card to display a dialog with an `ALERT` layout (see top of Figure 9-16). Note that this card shows a stack indicator at the top-right of the screen. The stack indicator provides feedback and suggests that users can access additional content if they tap on the card. When users tap on the card, `GlassAlertDialog` is displayed (see bottom of Figure 9-16).

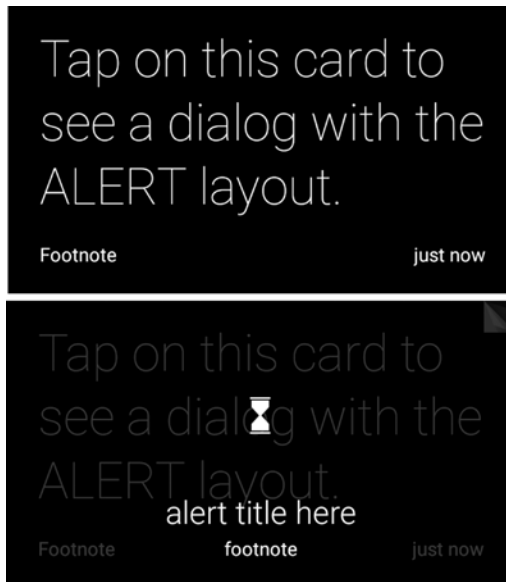


Figure 9-16. Demonstrating the use of the `ALERT` layout in a dialog. A card prompts users to tap on the touchpad to open the dialog (top). The dialog displays an `ALERT` layout on a translucent background (bottom)

1. Add a card to the end of the `CardScrollView` in `CardListActivity` and prompt users to tap on the card to display the dialog.

Showing the stack indicator of this card is as easy as calling its `showStackIndicator` method.

In the `initCards` method of `CardListActivity.java`:

```
mCards.add(new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText("Tap on this card to see a dialog with the ALERT layout.")
    .setFootnote("Footnote")
    .setTimestamp("just now")
    .showStackIndicator(true));
```

2. Start `GlassAlertDialog` when users tap on the card we created in step 1.

Modify the `CardScrollView`'s item click listener as follows.

In `CardListActivity.java`:

```
private AdapterView.OnItemClickListener mItemClickListener =
    new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        // last item clicked
        if (position == mCards.size() - 1) {
            GlassAlertDialog alertDialog = new GlassAlertDialog(CardListActivity.this,
                R.drawable.ic_timer_50, "alert title here", "footnote");

            alertDialog.setOnClickListener(new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    // no need to play Sounds.TAP. The dialog does it automatically.
                    Toast.makeText(CardListActivity.this, "Alert Dialog clicked",
                        Toast.LENGTH_SHORT).show();
                }
            });

            alertDialog.show();
        } else {
            AudioManager.playSoundEffect(Sounds.DISALLOWED);
        }
    }
};
```

Note that this listener plays the `DISALLOWED` sound when users click on any other card.

Try to use the card styles generated by `CardBuilder` whenever possible. `CardBuilder` not only saves you time, but it also makes your Glassware's appearance consistent with rest of the Glass platform.

The Ongoing Task Pattern

Unlike immersions (see Chapter 8), ongoing tasks are part of the timeline. When an ongoing task is running, a special kind of card called a LiveCard is inserted to the present and future section of the timeline (that is, to the left of the home screen). Unlike other cards, LiveCards can display animations, media, and other content that either periodically changes or contains motion. Users can scroll through the timeline and trigger voice commands without affecting existing LiveCards. All of the Google Now cards are examples of LiveCards, which are periodically updated to reflect the most recent information. In the case of the weather card, for instance, a background service periodically fetches the most recent forecast from the Internet and updates its corresponding LiveCard.

Use LiveCards to display information that doesn't require the user's constant attention (in which case immersions would be more appropriate). Since users can still access other timeline items when LiveCards are available, LiveCards are great at displaying content that users access relatively infrequently.

LiveCards Require Menus

Tapping on a LiveCard should always display an options menu. At the very least, this menu should contain an option to remove the LiveCard and stop its background service. Since LiveCards are updated from services, they cannot create an options menu directly. Instead, LiveCards create menus by starting an activity that displays an options menu. That is, when users select a menu item from a LiveCard, they are really selecting an option from an activity's menu.

Publishing LiveCards

Adding a LiveCard to the timeline is referred to as publishing. To publish a LiveCard, call its `publish` method, which takes a single parameter of type `LiveCard.PublishMode`. The publish mode specifies how a LiveCard is inserted into the timeline and can take two values:

- `PublishMode.REVEAL`: takes the user to the LiveCard after displaying an animation that shows the LiveCard being inserted into the timeline.
- `PublishMode.SILENT`: adds the LiveCard into the timeline without providing and visual feedback.

Updating LiveCards

LiveCards update their content from a background service in one of two ways:

- low-frequency rendering utilizes `RemoteViews` to periodically make changes, and
- high-frequency rendering lets you draw directly on a LiveCard's backing surface (think `SurfaceView`).

Use low-frequency rendering when a LiveCard does not require frequent updates and uses a layout that can be created with RemoteViews. If a LiveCard needs to be updated more than once a second, consider using high-frequency rendering. High-frequency rendering does not rely on RemoteViews and lets you use most of the graphics framework.

Low-Frequency Rendering

Low-frequency rendering updates LiveCards by using RemoteViews and is appropriate for content that only requires infrequent updates, such as the weather updates of the Google Now Glassware.

Note RemoteViews is essentially a view that can be displayed in another process. This requirement limits the types of views that can be used in layouts for RemoteViews. To see a list of the supported views, see <http://developer.android.com/guide/topics/appwidgets/index.html#CreatingLayout>.

Use low-frequency rendering when

- a LiveCard does not need updates at a rate faster than once a second, and
- the content of a LiveCard only requires views that are supported by RemoteViews.

This example demonstrates how to implement an ongoing task that displays the percentage of battery remaining on Glass with a LiveCard that uses low-frequency rendering.

Implementing Battery Service

All LiveCards are created from background services. In this case, BatteryService, which is started from a voice command, creates a LiveCard and updates its content.

1. Create a voice trigger with a keyword of “Show Card List.”

In res > xml > battery_trigger.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Start Battery Monitor" />
```

2. Declare BatteryService to be started in response to the “Show Card List” voice command.

In `AndroidManifest.xml`:

```
<service android:name=".BatteryService"
    android:icon="@drawable/ic_launcher">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>
    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/battery_trigger" />
</service>
```

3. Create a layout.

This layout displays a large `TextView` in the middle of the screen.

In `res > layout > centertext.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/text"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:textSize="130px"
    android:gravity="center"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

4. Declare member variables.

In `BatteryService.java`:

```
public class BatteryService extends Service {
    private static final String LIVE_CARD_TAG = "BatteryCard";
    private LiveCard mLiveCard;
    private RemoteViews mRemoteViews;
    ...
}
```

5. When the service is started, create a `LiveCard`.

Note that the `LiveCard` constructor's second parameter is a tag, which is a string that identifies the `LiveCard` for debugging purposes. After instantiating the `LiveCard`, create a `PendingIntent` that starts `BatteryMenuActivity`, which displays the `LiveCard`'s menu. When starting this activity, use the `FLAG_ACTIVITY_NEW_TASK` flag to specify that it should run on a separate task. Since services don't run on tasks, they cannot start activities without this flag. The `FLAG_ACTIVITY_CLEAR_TASK` flag ensures that the menu starts in an empty task even if a user has previously accessed the menu. Pass this `PendingIntent` to the `LiveCard`'s `setAction` method.

In `BatteryService.java`:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    super.onStartCommand(intent, flags, startId);

    if(mLiveCard == null) {
        mLiveCard = new LiveCard(this, LIVE_CARD_TAG);

        // live cards must have a menu
        Intent menuIntent = new Intent(this, BatteryMenuActivity.class);
        menuIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
            Intent.FLAG_ACTIVITY_CLEAR_TASK);
        mLiveCard.setAction(PendingIntent.getActivity(this, 0, menuIntent, 0));
        mLiveCard.publish(LiveCard.PublishMode.REVEAL);

        // register battery receiver
        IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
        Intent batteryStatus = registerReceiver(mBatteryLevelReceiver, ifilter);

        mRemoteViews = new RemoteViews(getPackageName(), R.layout.centertext);
        updateRemoteViews(batteryStatus);
    }

    return START_STICKY;
}
```

At the end of the `onStartCommand` method, we register a `BroadcastReceiver` that receives battery and power information, and we call `updateRemoteViews`, which calculates the remaining battery percentage and displays it in the `LiveCard`.

6. Implement `updateRemoteViews`.

The level and scale values, which are given as extras in the `batteryStatus` intent, indicate the current battery level and the maximum battery level, respectively. To calculate the battery percentage, divide these values and multiply by 100.

After calculating the battery percentage, update the `TextView` in `RemoteViews` and call `LiveCard`'s `setViews` method. Note that any changes made to `RemoteViews` do not update a `LiveCard`'s content until you call the `setViews` method. In other words, you need to call `setViews` any time you update its content, even if you've already called `setViews` before.

```
private void updateRemoteViews(Intent batteryStatus) {
    int level = batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
    int scale = batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1);

    // get battery percentage and round to nearest integer
    int batteryPct = Math.round(level / (float)scale * 100);
    mRemoteViews.setTextViewText(R.id.text, batteryPct + "%");
    mLiveCard.setViews(mRemoteViews);
}
```

7. Implement BroadcastReceiver.

This BroadcastReceiver receives battery level updates.

In BatteryService.java:

```
private BroadcastReceiver mBatteryLevelReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        updateRemoteViews(intent);
    }
};
```

8. When the service is stopped, remove the LiveCard from the timeline by calling its unpublish method. Additionally, unregister the battery status BroadcastReceiver.

In BatteryService.java:

```
@Override
public void onDestroy() {
    if(mLiveCard != null && mLiveCard.isPublished()) {
        mLiveCard.unpublish();
        mLiveCard = null;

        unregisterReceiver(mBatteryLevelReceiver);
    }

    super.onDestroy();
}
```

9. Implement the required onBind method.

In BatteryService.java:

```
@Override
public IBinder onBind(Intent intent) {
    return null;
}
```

Creating a Menu for a LiveCard

As we saw above, a LiveCard's `setAction` method specifies a `PendingIntent` which starts an activity that displays a menu. This section implements the `BatteryMenuActivity`, which contains no content and automatically opens its options menu as soon as it's started. This menu contains a single option called "stop" that stops `BatteryService` and consequently removes the LiveCard from the timeline.

1. Create a style for the menu.

This style specifies that the activity has a translucent background. This background increases the contrast of the menu's title while still revealing the content of the LiveCard underneath.

In res ► values ► styles.xml:

```
<style name="MenuTheme" parent="@android:style/Theme.DeviceDefault">
  <item name="android:windowBackground">@android:color/transparent</item>
  <item name="android:colorBackgroundCacheHint">@null</item>
  <item name="android:windowIsTranslucent">true</item>
  <item name="android:windowAnimationStyle">@null</item>
</style>
```

2. Declare BatteryMenuActivity with the theme we created in the previous step.

In AndroidManifest.xml:

```
<activity
  android:name=".BatteryMenuActivity"
  android:theme="@style/MenuTheme" />
```

3. Create a menu resource with a single item called “Stop.”

In res ► menu ► battery.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/stop"
    android:title="Stop"
    android:icon="@drawable/ic_stop" />
</menu>
```

4. Open the menu as soon as the activity is started.

In BatteryMenuActivity.java:

```
@Override
public void onAttachedToWindow() {
  super.onAttachedToWindow();
  openOptionsMenu();
}
```

5. Create options menus with the battery.xml resource from step 3.

In BatteryMenuActivity.java:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.battery, menu);
  return true;
}
```

6. When the “Stop” menu item is selected, stop BatteryService.

Recall that the service's `onDestroy` method removes the `LiveCard` from the timeline, so stopping `BatteryService` also removes the `LiveCard`.

In `BatteryMenuActivity.java`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        case R.id.stop:
            stopService(new Intent(this, BatteryService.class));
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

7. After users tap on a menu item, automatically dismiss the menu so they can return to the `LiveCard`.

In `BatteryMenuActivity.java`:

```
@Override
public void onOptionsItemSelected(MenuItem menu) {
    finish();
}
```

When you run this example, you should see a `LiveCard` that indicates the percentage of battery remaining as illustrated by Figure 9-17.

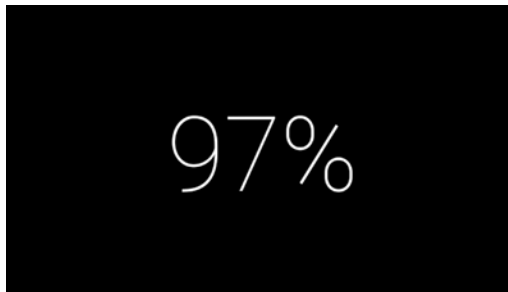


Figure 9-17. A `LiveCard` displays the percentage of battery remaining on Glass

`LiveCards` with low-frequency rendering are relatively simple to create, but high-frequency rendering is needed for more sophisticated applications.

High-Frequency Rendering

High-frequency rendering allows a service to draw content directly on a `LiveCard`. Unlike low-frequency rendering, high-frequency rendering does not rely on `RemoteViews`. Instead, a service that uses high-frequency rendering gives the `LiveCard` a callback that implements `DirectRenderingCallback`, which is able to draw content on the `LiveCard`.

Use high-frequency rendering when

- a LiveCard needs updates at a rate faster than once a second, or
- the content of a LiveCard cannot be implemented with RemoteViews.

This example demonstrates how to implement an ongoing task that displays a speedometer on a LiveCard that uses high-frequency rendering.

Implementing SpeedService

SpeedService, which is started from a voice command, creates a LiveCard and its DirectRenderingCallback.

1. Create a voice trigger with a keyword of “Start speedometer.”

In res ► xml ► speed_trigger.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Start speedometer" />
```

2. Declare SpeedService to be started in response to the “Start speedometer” voice command.

In AndroidManifest.xml:

```
<service android:name=".SpeedService"
    android:icon="@drawable/ic_launcher">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>
    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/speed_trigger" />
</service>
```

3. Declare member variables.

In SpeedService.java:

```
public class SpeedService extends Service {
    private static final String LIVE_CARD_TAG = "SpeedometerCard";
    private LiveCard mLiveCard;
    private SpeedRenderer mSpeedRenderer;
    ...
}
```

4. When the service is started, call the publishCard method, which creates a LiveCard and adds it to the timeline.

In `SpeedService.java`:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    publishCard();
    return START_STICKY;
}
```

5. When the service is stopped, remove the `LiveCard` from the timeline.

In `SpeedService.java`:

```
@Override
public void onDestroy() {
    unpublishCard();
    super.onDestroy();
}
```

6. In the `publishCard` method, instantiate the `LiveCard`, create a `PendingIntent` for its menu, and add an instance of `SpeedRenderer` as its `DirectRenderingCallback`.

Create the `LiveCard` and its menu just like in the previous example. Then, instantiate `SpeedRenderer`, which implements `DirectRenderingCallback` and is implemented in the next section. `DirectRenderingCallback` contains methods that draw directly on the `LiveCard`.

In `SpeedService.java`:

```
private void publishCard() {
    if(mLiveCard == null) {
        mLiveCard = new LiveCard(this, LIVE_CARD_TAG);

        // live cards must have a menu
        Intent menuIntent = new Intent(this, SpeedMenuActivity.class);
        menuIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
            Intent.FLAG_ACTIVITY_CLEAR_TASK);
        mLiveCard.setAction(PendingIntent.getActivity(this, 0, menuIntent, 0));

        mSpeedRenderer = new SpeedRenderer(this);
        mLiveCard.setDirectRenderingEnabled(true);
        mLiveCard.getSurfaceHolder().addCallback(mSpeedRenderer);
        mLiveCard.publish(LiveCard.PublishMode.REVEAL);
    }
}
```

7. In the `unpublishCard` method, remove the `LiveCard` from the timeline by calling its `unpublish` method.

In SpeedService.java:

```
private void unpublishCard() {
    if(mLiveCard != null) {
        mLiveCard.unpublish();
        mLiveCard = null;
    }
}
```

8. Implement the required onBind method.

In SpeedService.java:

```
@Override
public IBinder onBind(Intent intent) {
    return null;
}
```

Implementing SpeedMenuActivity

SpeedMenuActivity displays the LiveCard's options menu and is implemented very similarly to the menu from the previous example.

1. Declare SpeedMenuActivity.

This activity uses the theme called MenuTheme, which we implemented in the previous example and gives the menu a translucent background.

In AndroidManifest.xml:

```
<activity
    android:name=".SpeedMenuActivity"
    android:theme="@style/MenuTheme" />
```

2. Create a menu resource with one item called **stop**.

In res > menu > speed.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/stop"
        android:title="Stop"
        android:icon="@drawable/ic_stop" />
</menu>
```

3. Implement SpeedMenuActivity.

In `SpeedMenuActivity.java`:

```
public class SpeedMenuActivity extends Activity {
    @Override
    public void onAttachedToWindow() {
        super.onAttachedToWindow();
        openOptionsMenu();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.speed, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch(item.getItemId()) {
            case R.id.stop:
                stopService(new Intent(this, SpeedService.class));
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }

    @Override
    public void onOptionsItemSelected(MenuItem menu) {
        finish();
    }
}
```

Implement SpeedRenderer

`SpeedRenderer` implements `DirectRenderingCallback` and draws content directly on a `LiveCard`'s backing surface.

1. Declare member variables.

In `SpeedRenderer.java`:

```
public class SpeedRenderer implements DirectRenderingCallback {
    private static final long FRAME_TIME_MILLIS = 33;
    private static final float ANGLE_ZERO_MPH = -135f;
    private static final float DEGREES_PER_MPH = 9f/4f;
    private SurfaceHolder mHolder;
    private Bitmap mBackground, mNeedle;
    private Paint mBitmapPaint;
    private RenderThread mRenderThread;
    private boolean mPaused;
    ...
}
```

2. In the constructor, initialize the bitmaps that we'll use to draw the speedometer.

Additionally, initialize `mBitmapPaint`, which is used to draw the bitmaps with antialiasing enabled. Enabling this object's `filterBitmap` property ensures that anti aliasing is applied correctly.

In `SpeedRenderer.java`:

```
public SpeedRenderer(Context context) {
    mBackground = ((BitmapDrawable)context.getResources()
        .getDrawable(R.drawable.bg_speedometer)).getBitmap();
    mNeedle = ((BitmapDrawable)context.getResources()
        .getDrawable(R.drawable.needle)).getBitmap();
    mBitmapPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mBitmapPaint.setFilterBitmap(true);
}
```

3. Implement `renderingPaused`.

The `renderingPaused` method is part of `DirectRenderingCallback` and is called any time the `LiveCard` appears or leaves the screen. When the `LiveCard` is not visible, either because the screen is asleep or viewing something else, there's no need to draw content on the `LiveCard`. This second parameter of this callback method indicates whether we should pause rendering or not. The `mPaused` member variable stores this information and is used by the `updateRendering` to start or stop drawing on the `LiveCard`.

In `SpeedRenderer.java`:

```
@Override
public void renderingPaused(SurfaceHolder surfaceHolder, boolean paused) {
    mPaused = paused;
    updateRendering();
}
```

4. Implement `surfaceCreated`.

This callback method is part of `DirectRenderingCallback` and indicates when the backing surface of the `LiveCard` is available for drawing and provides access to its `SurfaceHolder`.

In `SpeedRenderer.java`:

```
@Override
public void surfaceCreated(SurfaceHolder holder) {
    mHolder = holder;
    updateRendering();
}
```

5. Implement `surfaceChanged`.

This method indicates the dimensions of the screen available for rendering. In this example, we won't use this information since we know the screen is 640x360, but we still implement the method since it's part of `DirectRenderingCallback`.

In `SpeedRenderer.java`:

```
@Override
public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {
}
```

6. Implement `surfaceDestroyed`.

This callback method is part of `DirectRenderingCallback` and indicates that the backing surface of the `LiveCard` is no longer available for drawing.

In `SpeedRenderer.java`:

```
@Override
public void surfaceDestroyed(SurfaceHolder holder) {
    mHolder = null;
    updateRendering();
}
```

7. Implement the `updateRendering` method and have it start and stop `RenderThread` as needed.

This method checks to see if the `LiveCard`'s `SurfaceHolder` is available and whether rendering is paused. If the `SurfaceHolder` is available and rendering is not paused, this method ensures that `RenderThread` is started. Otherwise, it ensures that `RenderThread` is stopped. `RenderThread` is responsible for drawing on the `LiveCard` as we'll see in the next section.

In `SpeedRenderer.java`:

```
private void updateRendering() {
    boolean shouldRender = (mHolder != null) && !mPaused;
    boolean rendering = mRenderThread != null;

    if (shouldRender != rendering) {
        if (shouldRender) {
            mRenderThread = new RenderThread();
            mRenderThread.start();
        } else {
            mRenderThread.quit();
            mRenderThread = null;
        }
    }
}
```

8. Implement the draw method.

This method is called by `RenderThread` and draws content on the `LiveCard`.

In `SpeedRenderer.java`:

```
private void draw() {
    Canvas
    canvas = mHolder.lockCanvas();

    if (canvas != null) {
        // draw background
        canvas.drawColor(Color.BLACK);
        canvas.drawBitmap(mBackground, 0, 0, mBitmapPaint);

        // calculate angle
        float angle = ANGLE_ZERO_MPH + DEGREES_PER_MPH * speedMph;

        // draw needle
        canvas.rotate(angle, 320, 196);
        canvas.drawBitmap(mNeedle, 315, 109, mBitmapPaint);

        mHolder.unlockCanvasAndPost(canvas);
    }
}
```

The `Canvas` class provides methods to draw shapes and images on the `LiveCard`. The canvas is obtained from `SurfaceHolder` with a `lockCanvas` call. Locking the canvas ensures that no other thread can access the canvas while `RenderThread` is drawing on it.

The code that uses the canvas seems complicated, but let's break it down:

- `canvas.drawColor(Color.BLACK)`: Fills the entire canvas with black.
- `canvas.drawBitmap(mBackground, 0, 0, mBitmapPaint)`: Draws the background of the speedometer on the `LiveCard`. The needle or pointer of the speedometer is not yet displayed.
- `float angle = ANGLE_ZERO_MPH + DEGREES_PER_MPH * speedMph`: Calculates the angle that the needle should be rotated to hit the appropriate speed. An angle of zero implies that the needle is directed straight up, as shown in Figure 9-18. When the speed is zero, the needle should be rotated by `ANGLE_ZERO_MPH` degrees clockwise, which is `-135` (note the negative sign). Since 90 degrees corresponds to 40 mph, every mph corresponds to $9/4$ degrees, which is the value of `DEGREES_PER_MPH`. Thus, a speed of `speedMph` corresponds to `DEGREES_PER_MPH*speedMph` from the angle at which speed is zero.
- `canvas.rotate(angle, 320, 196)`: Applies the rotation we previously calculated. The coordinates (320, 196) are the center of the needle, and specifying these values to the rotate command indicates that rotation should occur about this location.
- `canvas.drawBitmap(mNeedle, 315, 109, mBitmapPaint)`: Draws the needle on the `LiveCard`. The coordinates of (315, 109) are the location of the top-left corner of the needle bitmap when it's drawn in its initial position (that is, the position shown in Figure 9-18).

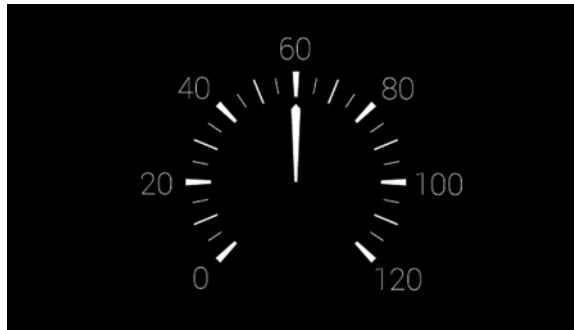


Figure 9-18. The speedometer's needle is directed straight up before it's rotated

Where did all the funky coordinates such as (315, 109) come from? I measured them directly from the image that I made for the background.

Implementing RenderThread

RenderThread is managed by SpeedRenderer and runs only when rendering is not paused. RenderThread is an inner class of SpeedRenderer and repeatedly calls the draw method we just implemented.

1. Declare member variables.

In SpeedRenderer.java:

```
private class RenderThread extends Thread {
    private AtomicBoolean mKeepRunning;
    private float mSpeed, mDeltaSpeed;

    public RenderThread() {
        mKeepRunning = new AtomicBoolean();
        mSpeed = 0;
        mDeltaSpeed = 0.5f;
    }
    ...
}
```

2. Implement the quit method, which stops the thread if it's running.

The mKeepRunning variable is an AtomicBoolean, which is essentially a thread safe boolean.

In SpeedRenderer.java:

```
public void quit() {
    mKeepRunning.set(false);
}
```

3. Repeatedly change the speed value and call the draw method to draw on the LiveCard.

In a real application, the speed could be obtained from GPS. In this demonstration, however, we'll generate values of speed that range from 0 to 110 mph. The run method updates the speed, calls the draw method, and sleeps for 33 milliseconds, which corresponds to a frame rate of about 30 frames per second. This method loops until `mKeepRunning` is set to false from the quit method.

In `SpeedRenderer.java`:

```
@Override
public void run() {
    mKeepRunning.set(true);

    while (mKeepRunning.get()) {
        // change speed values between 0 and 110 mph
        mSpeed += mDeltaSpeed;

        if(mSpeed > 110f) {
            mSpeed = 110;
            mDeltaSpeed = -mDeltaSpeed;
        }

        if(mSpeed < 0) {
            mSpeed = 0;
            mDeltaSpeed = -mDeltaSpeed;
        }

        draw(mSpeed);
        SystemClock.sleep(FRAME_TIME_MILLIS);
    }
}
```

At this point, run the “Start speedometer” voice command and you should see a `LiveCard` appear with a speedometer.

Implementing a Digital Speedometer

The previous example demonstrated how to create a `LiveCard` with high-frequency rendering. There are two aspects of `LiveCards` that were not illustrated by this example:

- How do you draw an existing view on a `LiveCard` with high-frequency rendering?
- How do you implement menu items that are more complex than the “Stop” item?

This example addresses both of these questions by showing how to implement a digital speedometer. Initially, the speedometer shows the current speed in miles per hour. The menu contains an item called “Use km/h” which changes the units of the speed into km/h (see Figure 9-19). When using km/h, the menu item that was formerly called “Use km/h” is called “Use mph” and changes the units back to mph.



Figure 9-19. The digital speedometer Glassware creates a LiveCard that displays the current speed. By default, the speed is displayed in miles per hour (left). Users can toggle the display to use kilometers per hour (right)

Implementing DigitalSpeedView

This layout contains a TextView that is centered in its container. In a few sections, we'll draw this layout directly onto a LiveCard.

1. Declare and initialize member variables.

In DigitalSpeedView.java:

```
public class DigitalSpeedView extends FrameLayout {
    private static final float KMH_PER_MPH = 1.60934f;
    private TextView mSpeed;
    private boolean mUseMetric;
    private float mSpeedMph;

    public DigitalSpeedView(Context context) {
        super(context);
        init(context);
    }

    public DigitalSpeedView(Context context, AttributeSet attrs) {
        super(context, attrs);
        init(context);
    }

    public DigitalSpeedView(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
        init(context);
    }
    ...
}
```

2. Inflate the content of the layout.

Inflate the content of the centertext layout, which we implemented in the battery level example, into the layout. Then, obtain a reference the mSpeed TextView.

In DigitalSpeedView.java:

```
private void init(Context context) {
    LayoutInflater.from(context).inflate(R.layout.centertext, this);
    mSpeed = (TextView) findViewById(R.id.text);
}
```

3. Implement a setter that takes the current speed in miles per hour.

In `DigitalSpeedView.java`:

```
public void setSpeedMph(float speedMph) {
    if(mSpeedMph != speedMph) {
        mSpeedMph = speedMph;
        mSpeedMph = 16.18f;
        updateText();
    }
}
```

4. Implement a setter that determines whether the output is in miles per hour or kilometers per hour.

In `DigitalSpeedView.java`:

```
public void setUseMetric(boolean useMetric) {
    if(mUseMetric != useMetric) {
        mUseMetric = useMetric;
        updateText();
    }
}
```

5. Implement the `updateText` method, which is called when the `mSpeedMph` or `mUseMetric` variables are modified.

This method sets the appropriate text on the `TextView`.

In `DigitalSpeedView.java`:

```
private void updateText() {
    float speed;
    String units;
    if(mUseMetric) {
        speed = mSpeedMph * KMH_PER_MPH;
        units = "km/h";
    } else {
        speed = mSpeedMph;
        units = "mph";
    }

    mSpeed.setText(String.format(Locale.US, "%.0f %s", speed, units));
}
```

Implementing `DigitalSpeedMenuActivity`

This activity displays the menu for the `LiveCard` that is created in `DigitalSpeedService`, which we'll implement in the next section.

1. Declare the activity with the `MenuTheme` theme.

In `AndroidManifest.xml`:

```
<activity
    android:name=".DigitalSpeedMenuActivity"
    android:theme="@style/MenuTheme" />
```

2. Create a menu resource with two items called **stop** and **change units**.

The **change units** item does not need a title since we'll set it in `onPrepareOptionsMenu`.

In `res > menu > digital_speed.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/stop"
        android:title="Stop"
        android:icon="@drawable/ic_stop" />

    <item
        android:id="@+id/change_units"
        android:icon="@drawable/ic_stop" />
</menu>
```

3. Declare constants.

In `DigitalSpeedMenuActivity.java`:

```
public class DigitalSpeedMenuActivity extends Activity {
    public static final String EXTRA_USE_METRIC = "use_metric";
    ...
}
```

4. In `onPrepareOptionsMenu`, check whether the LiveCard is using mph or km/h. Change the label of the second menu item accordingly.

The `EXTRA_USE_METRIC` extra is a boolean passed into the activity by `DigitalSpeedService` and indicates whether the LiveCard is using metric (that is, km/h).

In `DigitalSpeedMenuActivity.java`:

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    super.onPrepareOptionsMenu(menu);
    boolean useMetric = getIntent().getBooleanExtra(EXTRA_USE_METRIC, false);
    if(useMetric) {
        menu.findItem(R.id.change_units).setTitle("Use mph");
    } else {
        menu.findItem(R.id.change_units).setTitle("Use Km/h");
    }

    return true;
}
```

- When a user taps on the **stop** menu item, stop the `DigitalSpeedService` service. When a user taps on the menu item to change units, regardless of whether it's "Use mph" or "Use km/h", start the `DigitalSpeedService` with an action of `ACTION_CHANGE_UNITS`.

Starting a service that's already started calls its `onStartCommand` method once again. By specifying the `ACTION_CHANGE_UNITS` action, we tell `DigitalSpeedService` that a user tapped on the **change units** button. When used in this way, `startService` is a misnomer since the service is already started. We're really just sending it a message.

In `DigitalSpeedMenuActivity.java`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        case R.id.stop:
            stopService(new Intent(this, DigitalSpeedService.class));
            return true;
        case R.id.change_units:
            Intent resetIntent = new Intent(this, DigitalSpeedService.class);
            resetIntent.setAction(DigitalSpeedService.ACTION_CHANGE_UNITS);
            startService(resetIntent);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

- The remaining methods of the activity are identical to previous examples.

In `DigitalSpeedMenuActivity.java`:

```
@Override
public void onAttachedToWindow() {
    super.onAttachedToWindow();
    openOptionsMenu();
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.digital_speed, menu);
    return true;
}

@Override
public void onOptionsItemSelected(MenuItem item) {
    finish();
}
```

Implementing DigitalSpeedService

This service creates the LiveCard and handles requests to change units between mph and km/h that are issued by DigitalSpeedMenuActivity.

1. Create a voice trigger with a keyword of “Start Digital Speedometer.”

In res ► xml ► digital_speed_trigger.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Start Digital Speedometer" />
```

2. Declare DigitalSpeedService to be started in response to the “Start Digital Speedometer” voice command.

In AndroidManifest.xml:

```
<service android:name=".DigitalSpeedService"
    android:icon="@drawable/ic_launcher">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>
    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/digital_speed_trigger" />
</service>
```

3. Declare constants and member variables.

In DigitalSpeedService.java:

```
public class DigitalSpeedService extends Service {
    private static final String LIVE_CARD_TAG = "DigitalSpeed";
    public static final String ACTION_CHANGE_UNITS =
        "com.ocdevelopers.androidwearables.glassuinessentials.action.CHANGE_UNITS";
    private LiveCard mLiveCard;
    private DigitalSpeedRenderer mSpeedRenderer;
    private boolean mUseMetric;
    ...
}
```

4. When onStartCommand is called, either create the LiveCard or change its units.

If the intent’s action is ACTION_CHANGE_UNITS, then the user just clicked on the **Use mph** or **Use km/h** menu item, which means we should toggle the units the LiveCard is currently using. If the ACTION_CHANGE_UNITS action is not present, then the service was started from a voice command, in which case we should publish the LiveCard.

In `DigitalSpeedService.java`:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    handleIntent(intent);
    return START_NOT_STICKY;
}

private void handleIntent(Intent intent) {
    String action = intent.getAction();
    if(ACTION_CHANGE_UNITS.equals(action)) {
        changeUnits();
    } else {
        publishCard();
    }
}
```

5. Forward requests to change units to `SpeedRenderer`, which we'll implement in the next section.

In `DigitalSpeedService.java`:

```
private void changeUnits() {
    mUseMetric = !mUseMetric;
    mSpeedRenderer.setUseMetric(mUseMetric);
    generateMenu();
}
```

6. Creating a menu is identical to the previous examples, with one exception. `DigitalSpeedMenuActivity` needs to know whether the `LiveCard` is using mph or km/h so it can display an appropriate title on a menu item. The `EXTRA_USE_METRIC` extra tells `DigitalSpeedMenuActivity` what units the `LiveCard` is using.

In `DigitalSpeedService.java`:

```
private void generateMenu() {
    // live cards must have a menu
    Intent menuIntent = new Intent(this, DigitalSpeedMenuActivity.class);
    menuIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
        Intent.FLAG_ACTIVITY_CLEAR_TASK);
    menuIntent.putExtra(DigitalSpeedMenuActivity.EXTRA_USE_METRIC, mUseMetric);
    mLiveCard.setAction(PendingIntent.getActivity(this, 0, menuIntent,
        PendingIntent.FLAG_CANCEL_CURRENT));
}
```

7. Implement the rest of the methods, which are nearly identical to the previous examples.

In `DigitalSpeedService.java`:

```

@Override
public void onDestroy() {
    unpublishCard();
    super.onDestroy();
}

private void publishCard() {
    if(mLiveCard == null) {
        mLiveCard = new LiveCard(this, LIVE_CARD_TAG);
        generateMenu();

        mSpeedRenderer = new DigitalSpeedRenderer(this);
        mLiveCard.setDirectRenderingEnabled(true);
        mLiveCard.getSurfaceHolder().addCallback(mSpeedRenderer);
        mLiveCard.publish(LiveCard.PublishMode.REVEAL);
    }
}

private void unpublishCard() {
    if(mLiveCard != null) {
        mLiveCard.unpublish();
        mLiveCard = null;
    }
}

@Override
public IBinder onBind(Intent intent) {
    return null;
}

```

Implementing `DigitalSpeedRenderer`

This class implements `DirectRenderingCallback` and draws an instance of `DigitalSpeedView` directly on a `LiveCard`.

1. Declare constants and member variables.

In `DigitalSpeedRenderer.java`:

```

public class DigitalSpeedRenderer implements DirectRenderingCallback {
    private static final long FRAME_TIME_MILLIS = 33;
    private SurfaceHolder mHolder;
    private boolean mPaused, mUseMetric;
    private RenderThread mRenderThread;
    private DigitalSpeedView mDigitalSpeedView;
    ...
}

```

2. Implement the following methods, which are nearly identical to the corresponding methods of the previous example.

In `DigitalSpeedRenderer.java`:

```
public DigitalSpeedRenderer(Context context) {
    mDigitalSpeedView = new DigitalSpeedView(context);
}

@Override
public void renderingPaused(SurfaceHolder surfaceHolder, boolean paused) {
    mPaused = paused;
    updateRendering();
}

@Override
public void surfaceCreated(SurfaceHolder holder) {
    mHolder = holder;
    updateRendering();
}

@Override
public void surfaceDestroyed(SurfaceHolder holder) {
    mHolder = null;
    updateRendering();
}

private void updateRendering() {
    boolean shouldRender = (mHolder != null) && !mPaused;
    boolean rendering = mRenderThread != null;

    if (shouldRender != rendering) {
        if (shouldRender) {
            mRenderThread = new RenderThread();
            mRenderThread.setUseMetric(mUseMetric);
            mRenderThread.start();
        } else {
            mRenderThread.quit();
            mRenderThread = null;
        }
    }
}
```

3. Implement `surfaceChanged`.

In an activity, Android automatically performs a measure and layout pass on a view before displaying it. Doing so allows the view to determine how much space it occupies within the parent's dimensions. In this example, we must perform measure and layout manually, and we do so in `surfaceChanged`. Without this code, `mDigitalSpeedView` would not know what dimensions to take.

In `DigitalSpeedRenderer.java`:

```
@Override
public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {
    // measure and layout the view to find its dimensions
    int measuredWidth = View.MeasureSpec.makeMeasureSpec(width, View.MeasureSpec.EXACTLY);
    int measuredHeight = View.MeasureSpec.makeMeasureSpec(height, View.MeasureSpec.EXACTLY);
    mDigitalSpeedView.measure(measuredWidth, measuredHeight);
    mDigitalSpeedView.layout(0, 0, mDigitalSpeedView.getMeasuredWidth(),
        mDigitalSpeedView.getMeasuredHeight());
}
```

4. Create a setter that determines whether the `LiveCard` displays mph or km/h.

This setter is called by `DigitalSpeedService` any time a user taps on the menu item that toggles between mph and km/h.

In `DigitalSpeedRenderer.java`:

```
public void setUseMetric(boolean useMetric) {
    mRenderThread.setUseMetric(useMetric);
    mUseMetric = useMetric;
}
```

5. In the `draw` method, obtain a reference to the `LiveCard`'s canvas and draw the content of `mDigitalSpeedView`.

In `DigitalSpeedRenderer.java`:

```
private void draw() {
    Canvas canvas = mHolder.lockCanvas();

    if (canvas != null) {
        canvas.drawColor(Color.BLACK);
        mDigitalSpeedView.draw(canvas);
        mHolder.unlockCanvasAndPost(canvas);
    }
}
```

Implementing `RenderThread`

`RenderThread` is very similar to the implementation from the previous example.

1. Declare and initialize member variables.

Note that `mKeepRunning` and `mUseMetric` are `AtomicBooleans` to ensure thread safety, since they may be updated from another thread. Also, this class is an inner class of `DigitalSpeedRenderer`.

In `DigitalSpeedRendererer.java`:

```
private class RenderThread extends Thread {
    private AtomicBoolean mKeepRunning, mUseMetric;
    private float mSpeed, mDeltaSpeed;

    public RenderThread() {
        mKeepRunning = new AtomicBoolean();
        mUseMetric = new AtomicBoolean();
        mSpeed = 0;
        mDeltaSpeed = 0.5f;
    }
    ...
}
```

2. Setting `mKeepRunning` to false causes this thread to stop running.

In `DigitalSpeedRendererer.java`:

```
public void quit() {
    mKeepRunning.set(false);
}
```

3. This property determines whether the `LiveCard` uses mph or km/h.

In `DigitalSpeedRendererer.java`:

```
public void setUseMetric(boolean useMetric) {
    mUseMetric.set(useMetric);
}
```

4. In the `run` method, generate a speed value and use it to draw `DigitalSpeedView` on the `LiveCard`.

The speed values are generated just like in the previous example. Note that, before drawing the `DigitalSpeedView` on the `LiveCard`, we update its `useMetric` and `speedMph` properties.

In `DigitalSpeedRendererer.java`:

```
@Override
public void run() {
    mKeepRunning.set(true);

    while (mKeepRunning.get()) {
        mSpeed += mDeltaSpeed;

        if(mSpeed > 110f) {
            mSpeed = 110;
            mDeltaSpeed = -mDeltaSpeed;
        }
    }
}
```



```

    if(mSpeed < 0) {
        mSpeed = 0;
        mDeltaSpeed = -mDeltaSpeed;
    }

    mDigitalSpeedView.setUseMetric(mUseMetric.get());
    mDigitalSpeedView.setSpeedMph(mSpeed);
    draw();
    SystemClock.sleep(FRAME_TIME_MILLIS);
}
}

```

You can now use the “Start digital speedometer” voice command to start this example. Try to toggle the units of the speedometer between mph and km/h.

Displaying Progress and Status with Slider

A Slider is a class that has four main purposes:

- **Indeterminate Progress:** Indeterminate progress provides feedback that something is happening in the background but that the time to completion is unknown (see Figure 9-20). For example, Glass displays an indeterminate progress slider when it’s trying to obtain the results of speech recognition.

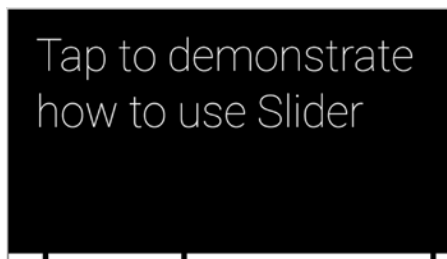


Figure 9-20. An indeterminate progress slider

- **Determinate Progress:** Determinate progress indicates the amount of time remaining for an event that occurs at a predictable time (see Figure 9-21). For instance, Glassware that asks users to respond to a question within 5 seconds could use a determinate progress slider to indicate how much time a user has to answer a question.

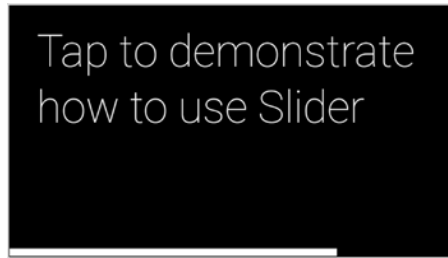


Figure 9-21. A determinate progress slider and a grace period slider have the same purpose, but a different intent

- **Grace Period:** In Glass, there are many actions that are tedious or impossible to undo, such as sending a message (see Figure 9-21). In these cases, Glass displays a grace period slider that gives users a certain amount of time to undo the action by swiping down. Try sending a message to see an example of a grace period slider.
- **Scroller:** Scroller sliders are used by `CardScrollView` to indicate the position of the current card relative to the total number of cards (see Figure 9-22).

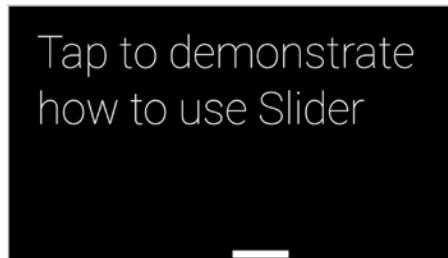


Figure 9-22. A scroller slider

This example demonstrates how to implement these four types of sliders.

Note The source code for this example is located in the app module of the `GlassUiEssentials` project. Run the module on the handheld device and start the “Slider demo” voice command.

1. Create a voice trigger with a keyword of “Slider demo.”

In res > xml > slider_trigger.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Slider demo" />
```

2. Declare `SliderActivity` to be started in response to the “Slider demo” voice command.

In `AndroidManifest.xml`:

```
<activity
    android:name=".SliderActivity"
    android:icon="@drawable/ic_launcher">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>
    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/slider_trigger" />
</activity>
```

3. Create a menu that contains four items, one for each type of slider.

In `res > menu > slider.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/demo_indeterminate"
        android:title="Indeterminate Slider"
        android:icon="@drawable/ic_launcher" />

    <item
        android:id="@+id/demo_determinate"
        android:title="Determinate Slider"
        android:icon="@drawable/ic_launcher" />

    <item
        android:id="@+id/demo_graceperiod"
        android:title="Grace Period Slider"
        android:icon="@drawable/ic_launcher" />

    <item
        android:id="@+id/demo_scroller"
        android:title="Scroller Slider"
        android:icon="@drawable/ic_launcher" />
</menu>
```

4. Declare member variables.

In `SliderActivity.java`:

```
public final class SliderActivity extends Activity {
    private static final int NUM_SCROLL_POSITIONS = 5;
    private Handler mHandler;
    private Slider mSlider;
    private Slider.Indeterminate mIndeterminate;
    private Slider.GracePeriod mGracePeriod;
    private int mPosition;
    private Slider.Scroller mScroller;
    private AudioManager mAudioManager;
    ...
}
```

5. Initialize member variables.

The `FLAG_KEEP_SCREEN_ON` flag ensures that the screen does not timeout.

With the GDK, a slider is a global component that is shared throughout all Glassware. We obtain a reference to slider with the `Slider.from` method, which takes a view as a parameter. We're able to control the slider only while this view is attached to the window that has focus.

In `SliderActivity.java`:

```
@Override
protected void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    mAudioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);

    View contentView = new CardBuilder(this, CardBuilder.Layout.TEXT)
        .setText("Tap to demonstrate how to use Slider")
        .getView();
    setContentView(contentView);
    mSlider = Slider.from(contentView);
    mHandler = new Handler();
}
```

6. Create an options menu.

In `SliderActivity.java`:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    getMenuInflater().inflate(R.menu.slider, menu);
    return true;
}
```

7. Open the options menu when a user taps on the touchpad.

In `SliderActivity.java`:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
        mAudioManager.playSoundEffect(Sounds.TAP);
        openOptionsMenu();
        return true;
    }
    return super.onKeyDown(keyCode, event);
}
```

8. When a menu item is selected, start the appropriate type of slider.

In `SliderActivity.java`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        case R.id.demo_indeterminate:
            demoIndeterminateSlider();
            return true;
        case R.id.demo_determinate:
            demoDeterminateSlider();
            return true;
        case R.id.demo_graceperiod:
            demoGracePeriodSlider();
            return true;
        case R.id.demo_scroller:
            demoScrollerSlider();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

9. Implement the `demoIndeterminateSlider` method and have it show an indeterminate slider that disappears after about 2500 milliseconds.

In `SliderActivity.java`:

```
private void demoIndeterminateSlider() {
    if (mIndeterminate == null) {
        mIndeterminate = mSlider.startIndeterminate();
        mHandler.postDelayed(mIndeterminateRunnable, 2500);
    }
}

private Runnable mIndeterminateRunnable = new Runnable() {
    @Override
    public void run() {
        mIndeterminate.hide();
        mIndeterminate = null;
    }
};
```

10. Implement the `demoDeterminateSlider` method and have it animate a determinate slider from its initial to its maximum position throughout 4 seconds.

The `mSlider.startDeterminate` method takes two parameters: the maximum and initial values of the position. If the maximum position is 20, for instance, a position of 10 occupies 50% of the slider.

In `SliderActivity.java`:

```
private void demoDeterminateSlider() {
    final Slider.Determinate determinate =
        mSlider.startDeterminate(100, 0);
    ObjectAnimator animator = ObjectAnimator.ofFloat(determinate, "position", 0, 100);

    animator.addListener(new AnimatorListenerAdapter() {
        @Override
        public void onAnimationEnd(Animator animation) {
            determinate.hide();
        }
    });

    animator.setDuration(4000)
        .start();
}
```

11. Implement the `demoGracePeriodSlider` method and have it play the SUCCESS sound when the grace period ends or the DISMISSED sound if the user dismisses it before it ends.

In `SliderActivity.java`:

```
private void demoGracePeriodSlider() {
    mGracePeriod = mSlider.startGracePeriod(mGracePeriodListener);
}

private final Slider.GracePeriod.Listener mGracePeriodListener =
    new Slider.GracePeriod.Listener() {
        @Override
        public void onGracePeriodEnd() {
            mAudioManager.playSoundEffect(Sounds.SUCCESS);
            mGracePeriod = null;
        }
        @Override
        public void onGracePeriodCancel() {
            mAudioManager.playSoundEffect(Sounds.DISMISSED);
            mGracePeriod = null;
        }
    };
};
```

12. The grace period slider does not automatically detect when a user swipes down to dismiss it. Instead, we must cancel the grace period manually.

In `SliderActivity.java`:

```
@Override
public void onBackPressed() {
    if (mGracePeriod != null) {
        mGracePeriod.cancel();
    } else {
        super.onBackPressed();
    }
}
```

13. Implement the `demoScrollerSlider`.

The `mSlider.startScroller` method contains two parameters: the maximum and initial position of the scroller slider. A `CardScrollView`, for instance, sets the maximum position to the index of the last item and the initial position to zero. In this example, we arbitrarily choose to display five slider positions.

In `SliderActivity.java`:

```
private void demoScrollerSlider() {
    if(mScroller == null) {
        mScroller = mSlider.startScroller(NUM_SCROLL_POSITIONS-1, 0);
        mPosition = 0;
        mScrollerRunnable.run();
    }
}

private Runnable mScrollerRunnable = new Runnable() {
    @Override
    public void run() {
        mScroller.setPosition(mPosition);
        ++mPosition;

        if(mPosition < NUM_SCROLL_POSITIONS) {
            mHandler.postDelayed(mScrollerRunnable, 800);
        } else {
            mScroller = null;
        }
    }
};
```

Note that a scroller slider hides after a brief period of inactivity, so there is no need to explicitly hide or remove it.

When you run `SliderActivity`, you should be able to see all four types of sliders. Keep in mind that the determinate progress and the grace period sliders look very similar, but their intent is different: determinate progress sliders show how much time is remaining for something that happens at a predictable moment while grace period sliders give users a few seconds to cancel an action such as sending a message.

Summary

We started this chapter by learning how to style cards that are consistent with the Glass platform by using `CardBuilder`. Then, we learned to implement ongoing tasks with `LiveCards` that use low-frequency rendering and high-frequency rendering. We implemented various examples of `LiveCards`, including a battery percentage indicator, a speedometer, and a digital speedometer. Finally, we learned to use the `Slider` class to display status and progress. In the next chapter, we'll learn to accept user input in several ways, including touchpad gestures, head gestures, and voice recognition.

Gesture and Voice Recognition

The primary ways of interacting with Glass are the touchpad and voice recognition. In this chapter, we'll learn to detect gestures on the touchpad—including tapping, swiping, and long pressing—and we'll learn several ways of leveraging voice recognition. Additionally, we'll learn to implement a less common way of interacting with Glass, head gestures. Let's get started.

Gestures on the Touchpad

The Android SDK supports all sorts of input devices, including touch screens and trackballs. When users interact with an input device, Android generates a series of events that describe the entire interaction at a low level. All of these events are typically passed into a callback as a `MotionEvent` object, which describes, for instance, how many fingers the user has on the touch screen and at what coordinates. An activity calls the `onTouchEvent(MotionEvent)` or `onTrackballEvent(MotionEvent)` methods when users interact with a touch screen or a trackball, respectively.

The primary input device of Glass is the long and narrow touchpad located at the side of the head. Android calls `onGenericMotionEvent(MotionEvent)` when users interact with the touchpad. To clarify, touchpads do not call `onTouchEvent(MotionEvent)` or `onTrackballEvent(MotionEvent)`, since these callback methods are reserved for touch screens and trackballs, respectively. Although touchpads and touch screens both have the word “touch” in them, users tap on specific parts of the screen with touch screens while touchpads can only detect relative motion, so these two devices must be processed in different ways.

Viewing Generic Motion Events

This example displays generic motion events, which are generated when a user interacts with the touchpad.

Note The source code for this example is located in the app module of the GestureAndVoice project. Run the module on Glass and start the “Demo motion events” voice command.

Begin by creating an activity called `MotionEventActivity` that is launched with the “demo motion events” voice command.

1. Create a voice trigger with the “Demo motion events” keyword.

In res ► xml ► `motionevent_trigger.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Demo motion events" />
```

2. Declare `MotionEventActivity` and have it launch in response to the voice command defined in the previous step.

In `AndroidManifest.xml`:

```
<activity android:name=".MotionEventActivity" >
  <intent-filter>
    <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
  </intent-filter>

  <meta-data
    android:name="com.google.android.glass.VoiceTrigger"
    android:resource="@xml/motionevent_trigger" />
</activity>
```

3. Create a layout for the activity.

In res ► layout ► `activity_motionevent.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:paddingTop="40px"
  android:paddingLeft="40px"
  android:useDefaultMargins="true"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
```

```
<TextView
    android:text="action:"
    android:fontFamily="sans-serif-thin"
    android:textSize="30px"
    android:textAllCaps="true"
    android:layout_gravity="right"
    android:layout_column="0"
    android:layout_row="0" />
```

```
<TextView
    android:id="@+id/action"
    android:text=""
    android:textSize="40px"
    android:fontFamily="sans-serif-light"
    android:minEms="10"
    android:layout_column="1"
    android:layout_row="0" />
```

```
<TextView
    android:text="X:"
    android:fontFamily="sans-serif-thin"
    android:textSize="30px"
    android:textAllCaps="true"
    android:layout_gravity="right"
    android:layout_column="0"
    android:layout_row="1" />
```

```
<TextView
    android:id="@+id/positionx"
    android:text=""
    android:textSize="40px"
    android:fontFamily="sans-serif-light"
    android:layout_column="1"
    android:layout_row="1" />
```

```
<TextView
    android:text="Y:"
    android:fontFamily="sans-serif-thin"
    android:textSize="30px"
    android:textAllCaps="true"
    android:layout_gravity="right"
    android:layout_column="0"
    android:layout_row="2" />
```

```
<TextView
    android:id="@+id/positiony"
    android:text=""
    android:textSize="40px"
    android:fontFamily="sans-serif-light"
    android:layout_column="1"
    android:layout_row="2" />
```

```

<TextView
    android:id="@+id/fingercount"
    tools:text="2 fingers detected"
    android:textSize="34px"
    android:fontFamily="sans-serif-thin"
    android:layout_columnSpan="2"
    android:layout_column="0"
    android:layout_row="3" />

```

```
</GridLayout>
```

4. Declare and initialize member variables.

In `MotionEventActivity.java`:

```

public class MotionEventActivity extends Activity {
    private TextView mFingerCount, mAction, mPositionX, mPositionY;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_motionevent);

        mFingerCount = (TextView) findViewById(R.id.fingercount);
        mAction = (TextView) findViewById(R.id.action);
        mPositionX = (TextView) findViewById(R.id.positionx);
        mPositionY = (TextView) findViewById(R.id.positiony);
    }
    ...

```

5. When a generic motion event is received, display its action, pointer count, and touch coordinates.

Note that the `event.getPointerCount` method, which returns the number of fingers that are on the touchpad, never returns zero. When users remove their last finger from the touchpad, this method returns a pointer count of 1 along with an `ACTION_UP` event.

In `MotionEventActivity.java`:

```

@Override
public boolean onGenericMotionEvent(MotionEvent event) {
    String action = "";

    int fingerCount = event.getPointerCount();
    switch(event.getActionMasked()) {
        case MotionEvent.ACTION_DOWN:
            action = "ACTION_DOWN";
            break;
        case MotionEvent.ACTION_MOVE:
            action = "ACTION_MOVE";
            break;
        case MotionEvent.ACTION_UP:
            action = "ACTION_UP";

```

```
        if(fingerCount == 1) {
            fingerCount = 0;
        }
        break;
    case MotionEvent.ACTION_CANCEL:
        action = "ACTION_CANCEL";
        break;
    default:
        return false;
}

mAction.setText(action);
mFingerCount.setText(fingerCount + " fingers on touchpad");
mPositionX.setText("" + event.getX());
mPositionY.setText("" + event.getY());

return super.onGenericMotionEvent(event);
}
```

The return value of `onGenericMotionEvent` is true if the motion event was handled and false otherwise. In this case, we return the value of `super.onGenericMotionEvent(event)` to ensure that the activity can be dismissed. Try returning true instead of the call to `super` and you'll see that the activity can no longer be dismissed by swiping down.

Run the application and try to see how the X and Y coordinates of the touchpad vary as you move your finger across it (see Figure 10-1). Try placing multiple fingers on the touchpad and seeing how the finger count changes in the display. Note that fingers that are located close to the front or the back of the touchpad are not detected and that the finger count seems to become less robust when there are three fingers on the touchpad. Moreover, Glass does not seem to support four or more fingers on the touchpad at a time.

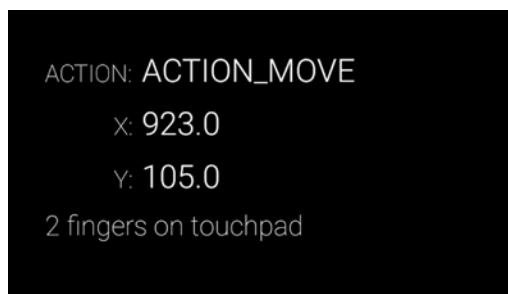


Figure 10-1. *MotionEventActivity* displays the action, coordinates, and finger count of the generic motion events that are generated when a user interacts with the touchpad

Using GestureDetector

The `onGenericMotionEvent(MotionEvent)` callback notifies an activity of how many fingers a user has on the touchpad and where they are located. This information can be used to figure out when a user performs gestures such as tapping or swiping on the touchpad. We could detect that the user performed a tap gesture, for instance, when the `onGenericMotionEvent(MotionEvent)` receives an `ACTION_UP` event since that implies a user touched and released the touchpad. Alas, this unsophisticated (and erroneous) implementation would detect a tap even when the user swipes on the touchpad. Although we could compensate by verifying that a user's finger does not move in between the `ACTION_DOWN` and `ACTION_UP` events, there could be additional corner cases to take into account. The implementation of more complex gestures would be significantly more tedious.

Fortunately, we don't need to figure out how to detect gestures from motion events since the GDK can do it for us with the `GestureDetector` class. Not only does `GestureDetector` spare us from having to write additional code to detect gestures, but it also ensures that our Glassware uses the same algorithms to detect gestures as the rest of the Glass platform. Consistency is essential, and gestures should occur under the exact same circumstances in any Glassware or users may get frustrated.

`GestureDetector` can notify you when a gesture occurs, when the number of fingers on the touchpad changes, and when the user swipes from side to side. More specifically, `GestureDetector` can detect the following gestures:

- `TAP`: a tap with a single finger
- `TWO_TAP`: a tap with two fingers
- `THREE_TAP`: a tap with three fingers
- `LONG_PRESS`: press and hold a single finger
- `TWO_LONG_PRESS`: press and hold two fingers
- `THREE_LONG_PRESS`: press and hold three fingers
- `SWIPE_UP`: swipe from down to up with a single finger
- `TWO_SWIPE_UP`: swipe from down to up with two fingers
- `SWIPE_RIGHT`: swipe from left to right with a single finger
- `TWO_SWIPE_RIGHT`: swipe from left to right with two fingers
- `SWIPE_DOWN`: swipe from up to down with a single finger
- `TWO_SWIPE_DOWN`: swipe from up to down with two fingers
- `SWIPE_LEFT`: swipe from right to left with a single finger
- `TWO_SWIPE_LEFT`: swipe from right to left with two fingers

Note that the `SWIPE_DOWN` gesture typically dismisses the current screen, and the `TWO_SWIPE_DOWN` gesture dismisses all activities, returns to the timeline, and puts the screen of Glass to sleep. Thus, Glassware should avoid using these two gestures for other purposes.

Detecting Gestures and Counting Fingers

This example uses `GestureDetector` to detect the gestures listed above and to display how many fingers the user has on the touchpad at any time.

We'll start by creating an activity that can be launched with the "demo touch gestures" voice command.

Note The source code for this example is located in the app module of the `GestureAndVoice` project. Run the module on Glass and start the "Demo touch gestures" voice command.

1. Create a voice trigger with the "Demo touch gestures" keyword.

In `res > xml > gesture_trigger.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword=" Demo touch gestures" />
```

2. Declare `GestureActivity` and have it launch in response to the voice command defined in the previous step.

In `AndroidManifest.xml`:

```
<activity android:name=".GestureActivity" >
  <intent-filter>
    <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
  </intent-filter>

  <meta-data
    android:name="com.google.android.glass.VoiceTrigger"
    android:resource="@xml/gesture_trigger" />
</activity>
```

3. Declare member variables.

In `GestureActivity.java`:

```
public class GestureActivity extends Activity {
  private GestureDetector mGestureDetector;
  private CardBuilder mCardBuilder;
  ...
}
```

4. In the `onCreate` method, initialize both the `GestureDetector` and the `CardBuilder`.

In `GestureActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mGestureDetector = new GestureDetector(this);
    mGestureDetector.setBaseListener(mBaseListener);
    mGestureDetector.setFingerListener(mFingerListener);

    mCardBuilder = new CardBuilder(this, CardBuilder.Layout.TEXT);
    updateText("Listening for gestures...");
}
```

5. `GestureDetector` needs access to the touchpad's `MotionEvent`s. Override the activity's `onGenericMotionEvent` and forward the `MotionEvent`s to `GestureDetector`:

This method should return `true` if the motion event is handled by either `super.onGenericMotionEvent` or by `GestureDetector`'s `onMotionEvent`.

In `GestureActivity.java`:

```
@Override
public boolean onGenericMotionEvent(MotionEvent event) {
    return super.onGenericMotionEvent(event) || mGestureDetector.onMotionEvent(event);
}
```

6. Every time `CardBuilder` is updated, we should call `setContentView`. Otherwise, the content of the activity will not update.

In `GestureActivity.java`:

```
private void updateText(String text) {
    mCardBuilder.setText(text);
    setContentView(mCardBuilder.getView());
}
```

7. Implement `GestureDetector.BaseListener`, which is a callback that `GestureDetector` calls every time it detects a gesture.

The `onGesture` method returns `true` if it handles a gesture. Gestures that are not handled here are handled by the activity. We intentionally do not handle the `SWIPE_DOWN` gesture so the activity can process it and dismiss the app in response to this gesture.

Also note that the `TWO_SWIPE_DOWN` gesture is a system gesture that dismisses all activities, returns to the timeline, and puts Glass to sleep. Thus, `GestureDetector` cannot detect this gesture.

In `GestureActivity.java`:

```
private GestureDetector.Baselistener mBaselistener = new GestureDetector.Baselistener() {
    @Override
    public boolean onGesture(Gesture gesture) {
        switch(gesture) {
            case TAP:
                updateText("TAP detected");
                return true;
            case TWO_TAP:
                updateText("TWO_TAP detected");
                return true;
            case THREE_TAP:
                updateText("THREE_TAP detected");
                return true;
            case LONG_PRESS:
                updateText("LONG_PRESS detected");
                return true;
            case TWO_LONG_PRESS:
                updateText("TWO_LONG_PRESS detected");
                return true;
            case THREE_LONG_PRESS:
                updateText("THREE_LONG_PRESS detected");
                return true;
            case SWIPE_UP:
                updateText("SWIPE_UP detected");
                return true;
            case TWO_SWIPE_UP:
                updateText("TWO_SWIPE_UP detected");
                return true;
            case SWIPE_RIGHT:
                updateText("SWIPE_RIGHT detected");
                return true;
            case TWO_SWIPE_RIGHT:
                updateText("TWO_SWIPE_RIGHT detected");
                return true;
            case SWIPE_DOWN:
                updateText("SWIPE_DOWN detected");
                // returning false allows the activity to process the gesture
                // swiping down wouldn't dismiss the activity if we returned true
                return false;
            case TWO_SWIPE_DOWN:
                // this will never be called
                updateText("TWO_SWIPE_DOWN detected");
                return true;
            case SWIPE_LEFT:
                updateText("SWIPE_LEFT detected");
                return true;
            case TWO_SWIPE_LEFT:
                updateText("TWO_SWIPE_LEFT detected");
                return true;
        }
    }
};
```

```

        default:
            return false;
    }
}
};

```

8. Implement `FingerListener`, which is a callback that `GestureDetector` calls any time it detects a change in the number of fingers that are on the touchpad.

In `GestureActivity.java`:

```

private GestureDetector.FingerListener mFingerListener = new GestureDetector.
FingerListener() {
    @Override
    public void onFingerCountChanged(int previousCount, int currentCount) {
        mCardBuilder.setFootnote(currentCount + " fingers on touchpad");
        setContentView(mCardBuilder.getView());
    }
};

```

Run this example to see the names of the gestures that are detected as you manipulate the touchpad (see Figure 10-2).

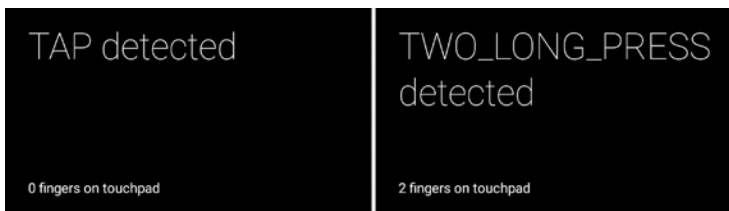


Figure 10-2. Viewing the gestures detected by `GestureDetector`

Detecting Scrolling on the Touchpad

The previous example shows how to use `GestureDetector`'s `BaseListener` and `FingerListener` to detect gestures (such as taps and swipes) and determine the number of fingers on the touchpad. `GestureDetector` has an additional callback called `ScrollListener`, which lets us know how far and how fast a user scrolls along the touchpad. Scrolling consists of moving a finger back and forth along the touchpad in a motion similar to swiping. Unlike swiping, however, scrolling is a slower and more deliberate motion that typically controls an on-screen element throughout the entire gesture. For instance, scrolling is used by Glass's movie player to let users rewind and fast-forward throughout a movie.

This Glassware example uses `ScrollListener` to let users select a number by swiping back and forth. Scrolling towards the front increases the number and scrolling towards the back decreases the number. The number increases or decreases proportionally to how fast and how long a user scrolls on the touchpad.

Note The source code for this example is located in the app module of the GestureAndVoice project. Run the module on Glass and start the “Demo scroll gestures” voice command.

1. Create a voice trigger with the “Demo scroll gestures” keyword.

In res ► xml ► scroll_trigger.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword=" Demo scroll gestures" />
```

2. Declare FingerScrollActivity and have it launch in response to the voice command defined in the previous step.

In AndroidManifest.xml:

```
<activity android:name=".FingerScrollActivity" >
  <intent-filter>
    <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
  </intent-filter>

  <meta-data
    android:name="com.google.android.glass.VoiceTrigger"
    android:resource="@xml/scroll_trigger" />
</activity>
```

3. Create a layout.

This layout displays a scroll gesture’s displacement, delta, and velocity, as we’ll see shortly. Additionally, it displays a count, which is the number that we modify with scrolling.

In res ► layout ► activity_finger_scroll.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:padding="40px"
  android:useDefaultMargins="true"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
    android:text="displacement:"
    android:fontFamily="sans-serif-thin"
    android:textSize="30px"
    android:textAllCaps="true"
    android:layout_gravity="right"
    android:layout_column="0"
    android:layout_row="0" />
```

```
<TextView
    android:id="@+id/displacement"
    android:text="0"
    android:textSize="40px"
    android:fontFamily="sans-serif-light"
    android:minEms="5"
    android:layout_column="1"
    android:layout_row="0" />

<TextView
    android:text="delta:"
    android:fontFamily="sans-serif-thin"
    android:textSize="30px"
    android:textAllCaps="true"
    android:layout_gravity="right"
    android:layout_column="0"
    android:layout_row="1" />

<TextView
    android:id="@+id/delta"
    android:text="0"
    android:textSize="40px"
    android:fontFamily="sans-serif-light"
    android:layout_column="1"
    android:layout_row="1" />

<TextView
    android:text="velocity:"
    android:fontFamily="sans-serif-thin"
    android:textSize="30px"
    android:textAllCaps="true"
    android:layout_gravity="right"
    android:layout_column="0"
    android:layout_row="2" />

<TextView
    android:id="@+id/velocity"
    android:text="0"
    android:textSize="40px"
    android:fontFamily="sans-serif-light"
    android:layout_column="1"
    android:layout_row="2" />

<TextView
    android:id="@+id/count"
    tools:text="42"
    android:textSize="70px"
    android:layout_gravity="center_horizontal"
    android:layout_columnSpan="2"
    android:layout_column="0"
    android:layout_row="3" />

</GridLayout>
```

4. Declare constants and member variables.

In `FingerScrollActivity.java`:

```
public class FingerScrollActivity extends Activity {
    private static float FLING_VELOCITY_CUTOFF = 3f;
    private static float DECELERATION_CONSTANT = 0.2f;
    private static float COUNT_DAMPENER = 100f;
    private static float TIME_LENGTHENING = 12f;
    private float mReleaseVelocity;
    private ValueAnimator mInertialScrollAnimator;
    private GestureDetector mGestureDetector;
    private TextView mDisplacement, mDelta, mVelocity, mCountText;
    private float mCount, mPrevCount;
    private AudioManager mAudioManager;
    ...
}
```

5. Initialize member variables.

The `updateCount` method, which is called at the end of `onCreate`, displays the current value of `mCount` on the screen. We'll implement this method shortly.

In `FingerScrollActivity`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_finger_scroll);
    mDisplacement = (TextView) findViewById(R.id.displacement);
    mDelta = (TextView) findViewById(R.id.delta);
    mVelocity = (TextView) findViewById(R.id.velocity);
    mCountText = (TextView) findViewById(R.id.count);
    mAudioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);

    mGestureDetector = new GestureDetector(this);
    mGestureDetector.setScrollListener(mScrollListener);
    mGestureDetector.setFingerListener(mFingerListener);

    mCount = mPrevCount = 0;
    updateCount();
}
```

6. `GestureDetector` needs access to the touchpad's `MotionEvent`s. Override the activity's `onGenericMotionEvent` and forward the `MotionEvent`s to `GestureDetector`.

In `FingerScrollActivity`:

```
@Override
public boolean onGenericMotionEvent(MotionEvent event) {
    return super.onGenericMotionEvent(event) || mGestureDetector.onMotionEvent(event);
}
```

7. Implement ScrollListener.

When a user moves one or more fingers across the touchpad, `GestureDetector` repeatedly calls the `onScroll` method of a `ScrollListener`. The `onScroll` method contains three parameters, all of which are floats:

- **displacement:** the difference between the current touchpad's X position and the X position of the first touch.
- **delta:** the difference between the current touchpad's position and the previous value.
- **velocity:** the speed of the finger scroll in touchpad-units per second.

In the `onScroll` method, we display the values of its three parameters on the screen by calling the `updateText` method so you can see how they change as you move a finger across the touchpad. Additionally, we update the value of the current count according to how much the user scrolls.

In `FingerScrollActivity.java`:

```
private GestureDetector.ScrollListener mScrollListener = new GestureDetector.  
ScrollListener() {  
    @Override  
    public boolean onScroll(float displacement, float delta, float velocity) {  
        mReleaseVelocity = velocity;  
  
        updateText(displacement, delta, velocity);  
        // Math.min(1, Math.abs(velocity)) finds the magnitude of the velocity  
        // and limits it to a maximum of 1. Multiplying this by delta increases  
        // the count proportionally to how far and how fast a user scrolls.  
        mCount += delta*Math.min(1, Math.abs(velocity)) / COUNT_DAMPENER;  
        updateCount();  
  
        return true;  
    }  
};
```

Note that `mCount` is declared as a float even though we are keeping track of an integer count. By doing so, we can modify `mCount` by a decimal value that is proportional to how far and how fast a user scrolls on the touchpad. The real count is then obtained by casting `mCount` to an integer. As a result, we modify the value of the count only once a user scrolls a certain amount.

Additionally, we want to make sure that the count updates at a reasonable rate. That is, we don't want a small swipe to increase the counter by some ridiculously large magnitude. Instead, we want a small swipe to increase the count by no more than 5 or 10. To do so, we divide the increase in count by the constant `COUNT_DAMPENER`, which is currently set to 100. Trial and error helps you pick a good value for this constant.

8. Implement `updateText`, which displays the values of displacement, delta, and velocity on the screen.

In `FingerScrollActivity.java`:

```
private void updateText(float displacement, float delta, float velocity) {
    mDisplacement.setText(String.format(Locale.US, "%.2f", displacement));
    mDelta.setText(String.format(Locale.US, "%.2f", delta));
    mVelocity.setText(String.format(Locale.US, "%.2f", velocity));
}
```

9. Implement `updateCount`, which casts `mCount` to an integer to obtain the real count and displays this value on the screen. Additionally, this method plays the TAP sound every time the count values changes.

In `FingerScrollActivity.java`:

```
private void updateCount() {
    mCountText.setText("" + (int)mCount);

    if((int)mPrevCount != (int)mCount) {
        mAudioManager.playSoundEffect(Sounds.TAP);
        mPrevCount = mCount;
    }
}
```

Note In this example, we're implementing `ScrollListener` to receive events from `GestureDetector`. `ScrollListener` receives events regardless of how many fingers are on the touchpad. In contrast, `OneFingerScrollListener` and `TwoFingerScrollListener` only receive events if there are exactly one or two fingers on the touchpad, respectively. To set one and two finger listeners, call `GestureDetector`'s `setOneFingerScrollListener` and `setTwoFingerScrollListener` instead of `setScrollListener`.

At this point, run the example to see the screen shown in Figure 10-3. Try scrolling at different speeds and observing the values of displacement, delta, and velocity that occur as a result.

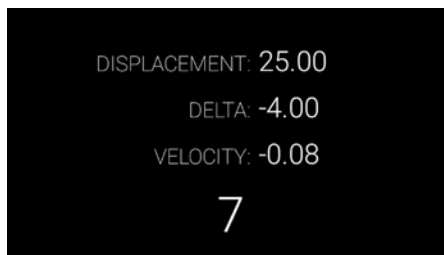


Figure 10-3. Viewing the `onScroll` method's parameters and implementing a counter

Implementing Inertial Scroll

The latter part of the example was adapted from the timer sample (see <https://github.com/googleglass/gdk-timer-sample>).

If you have installed the timer Glassware from MyGlass, start it with the “OK Glass... Start a timer” voice command and observe how you can scroll on the touchpad to set the duration for the timer. It behaves similarly to our example’s implementation with the exception of inertial scrolling. Normally, scrolling stops as soon as you remove a finger from the touchpad. With inertial scrolling, on the other hand, scrolling continues after you remove a finger and comes to a gradual stop.

Inertial scrolling isn’t essential, but it gives the Glassware a nice polished feel. Let’s add inertial scrolling to our Glassware. Scrolling quickly will increase or decrease the count for a little bit after you remove your finger from the touchpad.

Note The source code for this example is located in the app module of the GestureAndVoice project. Run the module on Glass and start the “Demo inertial scroll” voice command.

We implement inertial scrolling with a `ValueAnimator`, which repeatedly modifies a property between a range of values throughout a specified period of time. When started, `ValueAnimator` repeatedly calls a callback method with a float parameter that varies through the range of values.

In this example, we’ll start a `ValueAnimator` when a user releases the touchpad after scrolling faster than a threshold velocity. Then, `ValueAnimator` continues to increase or decrease the count for a brief moment.

1. Copy the contents of `FingerScrollActivity.java` into `InertialScrollActivity.java`.

The majority of the implementation of `InertialScrollActivity` is identical to `FingerScrollActivity`.

2. Declare additional constants and variables.

Add these declarations to `InertialScrollActivity.java`:

```
private static float FLING_VELOCITY_CUTOFF = 3f;
private static float DECELERATION_CONSTANT = 0.2f;
private static float TIME_LENGTHENING = 12f;
private float mReleaseVelocity;
```

3. Add a `FingerListener` to `GestureDetector` and initialize `ValueAnimator`.

Append the following code to the end of the `onCreate` method.

In `InertialScrollActivity.java`:

```
mGestureDetector.setFingerListener(mFingerListener);
// implement inertial score for the counter
mInertialScrollAnimator = new ValueAnimator();
mInertialScrollAnimator.setInterpolator(new DecelerateInterpolator());
mInertialScrollAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
    @Override
    public void onAnimationUpdate(ValueAnimator animation) {
        mCount = (Float) animation.getAnimatedValue();
        updateCount();
    }
});
```

Here are a few key points:

- A `DecelerateInterpolator` specifies that the rate of change of the animation starts out fast and then gradually decelerates. In this case, it makes the count increase or decrease at a faster rate at the beginning than at the end.
 - The `AnimatorUpdateListener` gets notified every time the animation changes value. The animation takes the value of `mCount` throughout a certain range. Calling the `updateCount` method updates the value of count displayed on the screen. Recall that the value of the count is really `mCount` casted to an integer.
4. Implement the `FingerListener`, which starts `ValueAnimator` when users remove their last finger from the touchpad after scrolling faster than a predefined threshold.

The `mReleaseVelocity` variable is updated from the `onScroll` method and contains the latest scroll velocity. If this velocity is greater than a threshold (that is, `FLING_VELOCITY_CUTOFF`), this code starts `ValueAnimator`.

To do so, we calculate the duration and the value at which the counter should settle assuming a constant rate of deceleration, which we pick through trial and error (see `DECELERATION_CONSTANT`). If you are unfamiliar with these formulas, which should be reminiscent of high school physics, simply keep in mind that they calculate the final value and duration of `mCount` if it were to increase or decrease while decelerating at a constant rate.

In `InertialScrollActivity.java`:

```
private GestureDetector.FingerListener mFingerListener = new GestureDetector.
FingerListener() {
    @Override
    public void onFingerCountChanged(int previousCount, int currentCount) {
        boolean wentDown = currentCount > previousCount;
```

```

// when the user releases the last finger
if (currentCount == 0 && !wentDown) {
    // Only start inertial scroll if the velocity is greater than the cutoff
    if (Math.abs(mReleaseVelocity) > FLING_VELOCITY_CUTOFF) {
        // calculate a range and a duration
        float deceleration = Math.signum(mReleaseVelocity) * -DECELERATION_CONSTANT;
        float flingTime = -mReleaseVelocity / deceleration * TIME_LENGTHENING;
        float totalDelta = mReleaseVelocity * mReleaseVelocity / 2f / -deceleration;

        mInertialScrollAnimator.cancel();
        mInertialScrollAnimator.setFloatValues(mCount, mCount + totalDelta/100);
        mInertialScrollAnimator.setDuration((long) flingTime);
        mInertialScrollAnimator.start();
    }
} else {
    mInertialScrollAnimator.cancel();
}
}
};

```

5. Cancel an animation if the user leaves the Glassware.

Note that calling the `ValueAnimator`'s `cancel` has no effect if there is no animation.

In `InertialScrollActivity.java`:

```

@Override
protected void onPause() {
    mInertialScrollAnimator.cancel();
    super.onPause();
}

```

Run the app and try scrolling at different speeds to see the effect of inertial scrolling.

Head Gestures

Glass can detect gestures made by a user's head just as it can detect gestures made on the touchpad. The main difference is that head gestures don't have an action that is comparable to tapping on the touchpad. That is, although we can "swipe" with a quick head nod in any direction, we can't perform any head gestures comparable to tapping. A head "swipe" gesture is referred to as a nod gesture.

Glass natively uses a head nod gesture to implement "head nudge," which turns the screen off when you perform a head nod similar to the "what's up" gesture (see Figure 10-4). You can enable or disable head nudge from the settings menu.

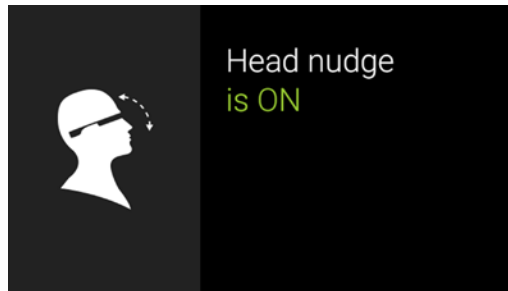


Figure 10-4. Enable or disable head nudge from the settings menu

Unfortunately, the GDK does not implement head nod detection as of XE22, but we can implement it ourselves by leveraging the gyroscopic sensors of Glass.

Gyroscopic sensors measure the angular velocity about three axes. The coordinate system of Glass is shown in Figure 10-5.

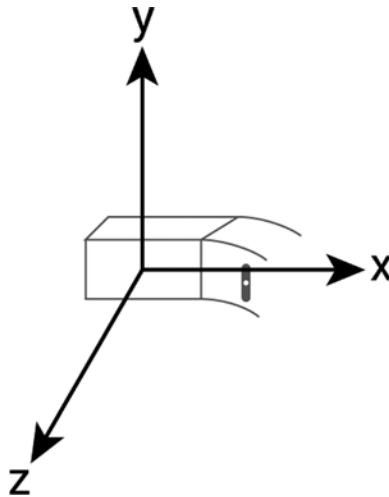


Figure 10-5. The coordinate system of Glass

The gyroscopic sensor measures how fast Glass is rotating about the X, Y, and Z axes, independently. As a user looks up, the gyroscopic sensor detects a positive rotation about the X axis. As a user looks down, the sensor detects a negative rotation about the X axis. Similarly, the sensor detects a positive or negative rotation about the Y axis as a user looks left or right, respectively.

Detecting Head Nods

NodDetector is an inner class of the HeadGestureDetector class. NodDetector is responsible for detecting nods in a single direction. Detecting nods in four directions (that is, up, down, left, and right) requires instantiating four NodDetectors, as we'll see shortly.

NodDetector has a single method called `addAngularVelocity` that returns true if a nod was detected and false otherwise. To detect a head nod, we can check if the angular velocity about one of these axes becomes sufficiently large and then becomes sufficiently small. If the angular velocity about an axis exceeds a certain threshold then a head nod is beginning to occur. Then, we wait for the angular velocity to drop low enough so that the user's head either stopped moving or is moving in the opposite direction. We check to see how long the user's head took between the initial start time and the time at which the velocity dropped, we detect a head nod. Otherwise, we assume that the user simply looked up or performed a head movement different from a head nod.

Note Head gestures suffer from the “Midas touch problem,” which occurs with interfaces in which there is no mechanism to disengage the input device. While users can let go of a touchpad when they have finished using it, their heads will always be moving, so natural head motions can be confused with gestures. In this example, we address the Midas touch problem by constraining head nods to quick movements.

In `HeadGestureDetector.java`:

```
private static class NodDetector {
    private static float THRESHOLD = 2f;
    private static float SMALL_THRESH = 0.1f;
    private static long MAX_DURATION = TimeUnit.MILLISECONDS.toNanos(1000);
    private long mStartTime = -1;

    public boolean addAngularVelocity(long timestamp, float angularVelocity) {
        if(mStartTime == -1 && angularVelocity > THRESHOLD) {
            // in nanoseconds
            mStartTime = timestamp;
        } else if(mStartTime > 0 && angularVelocity < SMALL_THRESH) {
            long deltaTime = timestamp - mStartTime;
            mStartTime = -1;
            if(deltaTime < MAX_DURATION) {
                return true;
            }
        }
        return false;
    }
}
```

Implementing HeadGestureDetector

HeadGestureDetector utilizes four NodDetectors to detect four gestures: nod up nod down, nod left, and nod right.

1. Create an interface for a callback that notifies when a nod has been detected in a particular direction.

In HeadGestureDetector.java:

```
public interface HeadGestureListener {
    void onNodUp();
    void onNodDown();
    void onNodLeft();
    void onNodRight();
}
```

2. Declare and initialize variables.

We create an array of four NodDetectors to detect four different gestures: nod up, nod down, nod left, and nod right.

In HeadGestureDetector.java:

```
private HeadGestureListener mHeadGestureListener;
private NodDetector[] mNodDetectors;

public HeadGestureDetector() {
    mNodDetectors = new NodDetector[4];
    for(int i=0; i<mNodDetectors.length; ++i) {
        mNodDetectors[i] = new NodDetector();
    }
}
```

3. Create a getter and a setter for the HeadGestureListener, which notifies the client class when a head nod was detected.

In HeadGestureDetector.java:

```
public HeadGestureListener getHeadGestureListener() {
    return mHeadGestureListener;
}

public void setHeadGestureListener(HeadGestureListener listener) {
    mHeadGestureListener = listener;
}
```

4. Implement the `onSensorChanged` method, which is called by the client class any time the gyroscopic sensor receives new input.

Looking up corresponds to rotation about the X axis. The `onSensorChanged` method calls the `detectNod` method with a timestamp and the value of the rotation about the X axis. The `detectNod` method returns true if it detects a nod, in which case we call the `triggerNod` method to trigger the appropriate method of the `HeadGestureListener`. The `onSensorChanged` method repeats this procedure for the nod down, nod left, and nod right gestures, which correspond to rotation about the negative X axis, Y axis, and negative Y axis, respectively.

In `HeadGestureDetector.java`:

```
public void onSensorChanged(SensorEvent sensorEvent) {
    if(sensorEvent.sensor.getType() != Sensor.TYPE_GYROSCOPE) {
        return;
    }
    if (sensorEvent.accuracy == SensorManager.SENSOR_STATUS_UNRELIABLE) {
        return;
    }

    // values in radians per second
    float axisX = sensorEvent.values[0];
    float axisY = sensorEvent.values[1];
    float axisZ = sensorEvent.values[2];

    // nod up, nod down, nod left, and nod right, respectively
    float[] values = new float[] { axisX, -axisX, axisY, -axisY };

    for(int index = 0; index<4; ++index) {
        if (detectNod(index, sensorEvent.timestamp, values[index])) {
            triggerNod(index);
        }
    }
}
```

5. Implement the `detectNod` method.

This method uses the appropriate `NodDetector` to figure out if a nod was detected, in which case it returns true.

In `HeadGestureDetector.java`:

```
private boolean detectNod(int index, long timestamp, float angularVelocity) {
    if(mNodDetectors[index].addAngularVelocity(timestamp, angularVelocity)) {
        if (mHeadGestureListener != null) {
            return true;
        }
    }
    return false;
}
```

6. Implement `triggerNod`, which triggers the appropriate method of `HeadGestureListener` once a nod is detected.

In `HeadGestureDetector.java`:

```
private void triggerNod(int index) {
    switch(index) {
        case 0:
            mHeadGestureListener.onNodUp();
            break;
        case 1:
            mHeadGestureListener.onNodDown();
            break;
        case 2:
            mHeadGestureListener.onNodLeft();
            break;
        case 3:
            mHeadGestureListener.onNodRight();
            break;
    }
}
```

Implementing HeadGestureActivity

This activity utilizes `HeadGestureDetector` and displays a message that indicates when a user nods in any direction.

1. Create a voice trigger with the “Demo head gestures” keyword.

In `res > xml > headgesture_trigger.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword=" Demo head gestures" />
```

2. Declare `HeadGestureActivity` and have it launch in response to the voice command defined in the previous step.

In `AndroidManifest.xml`:

```
<activity android:name=".HeadGestureActivity" >
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>

    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/headgesture_trigger" />
</activity>
```

3. Declare member variables.

In `HeadGestureActivity.java`:

```
public class HeadGestureActivity extends Activity {
    private HeadGestureDetector mHeadGestureDetector;
    private CardBuilder mCardBuilder;
    private SensorManager mSensorManager;
    private Sensor mSensor;
    ...
}
```

4. Implement the `onCreate` method.

The `onCreate` method;

1. displays a message that says “Listening for head gestures...”
2. initializes a gyroscopic sensor, and
3. initializes a `HeadGestureDetector`.

This method also adds two flags: `FLAG_KEEP_SCREEN_ON` ensures that the screen doesn’t timeout and go to sleep, and `FLAG_DISABLE_HEAD_GESTURES` disables “head nudge” when this activity is in focus. Recall that “head nudge” turns off the screen when a user performs a nod up gesture. This activity displays a message every time a nod gesture is detected, and having the screen turn off would be counter productive.

In `HeadGestureActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    getWindow().addFlags(WindowUtils.FLAG_DISABLE_HEAD_GESTURES);
    super.onCreate(savedInstanceState);

    mCardBuilder = new CardBuilder(this, CardBuilder.Layout.TEXT);
    updateText("Listening for head gestures...");

    mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
    if(mSensor == null) {
        Toast.makeText(this, "Rotation vector sensor required.", Toast.LENGTH_SHORT).show();
        finish();
    }

    mHeadGestureDetector = new HeadGestureDetector();
    mHeadGestureDetector.setHeadGestureListener(mHeadGestureListener);
}
```


5. Register and unregister for sensor updates in `onResume` and `onPause`, respectively.

We want to receive updates from the gyroscopic sensor only when the activity is in the foreground.

In `HeadGestureActivity.java`:

```
@Override
protected void onResume() {
    super.onResume();
    mSensorManager.registerListener(mSensorEventListener, mSensor, SensorManager.SENSOR_
    DELAY_UI);
}

@Override
protected void onPause() {
    mSensorManager.unregisterListener(mSensorEventListener);
    super.onPause();
}
```

6. Implement `updateText`, which displays text on the screen.

As usual, we must call `setContentView` after modifying a `CardBuilder`.

In `HeadGestureActivity.java`:

```
private void updateText(String text) {
    mCardBuilder.setText(text);
    setContentView(mCardBuilder.getView());
}
```

7. Forward sensor updates to `HeadGestureDetector`.

In `HeadGestureActivity.java`:

```
private SensorEventListener mSensorEventListener = new SensorEventListener() {
    @Override
    public void onSensorChanged(SensorEvent event) {
        mHeadGestureDetector.onSensorChanged(event);
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
    }
};
```

8. Output a message when a head nod is detected.

In `HeadGestureActivity.java`:

```
private HeadGestureDetector.HeadGestureListener mHeadGestureListener =
    new HeadGestureDetector.HeadGestureListener() {
    @Override
    public void onNodUp() {
        updateText("nod up detected");
    }

    @Override
    public void onNodDown() {
        updateText("nod down detected");
    }

    @Override
    public void onNodLeft() {
        updateText("nod left detected");
    }

    @Override
    public void onNodRight() {
        updateText("nod right detected");
    }
};
```

When you run the example, it should display a message any time it detects a head nod (see Figure 10-6).

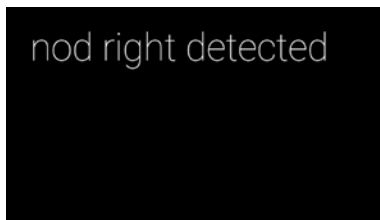


Figure 10-6. This example Glassware outputs a message every time it detects a head nod

Voice Recognition

So far, we've used voice recognition to start immersions and ongoing tasks from the voice command menu. In this section, we'll learn to use voice recognition to request text from users and to let them select menu items without relying on the touch screen.

Voice Command Prompts

Voice commands from the “OK Glass” menu can request additional voice input from the user. This example demonstrates how to implement voice command prompts.

1. Create a voice trigger with the “Demo voice recognition” keyword.

In res > xml > voicerec_trigger.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Demo voice recognition">
  <input prompt="say something" />
</trigger>
```

Note that, unlike any example we’ve seen so far, the trigger node has a child called input. The input node specifies that the voice command should prompt the user for additional text. The prompt attribute specifies what message the system should display while prompting the user for input. The above trigger will generate the prompt shown in Figure 10-7.

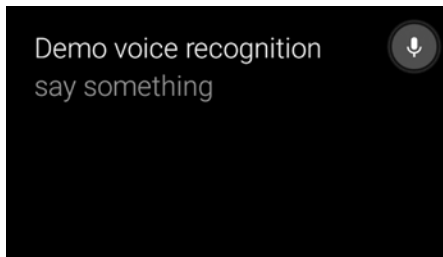


Figure 10-7. A voice command prompt requests additional input

2. Declare `VoiceRecognitionActivity` and have it launch in response to the voice command defined in the previous step.

In `AndroidManifest.xml`:

```
<activity android:name=".VoiceRecognitionActivity">
  <intent-filter>
    <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
  </intent-filter>

  <meta-data
    android:name="com.google.android.glass.VoiceTrigger"
    android:resource="@xml/voicerec_trigger" />
</activity>
```

3. Implement VoiceRecognitionActivity.

The activity retrieves the results of the voice command prompt as a list of strings stored in an intent extra. The list contains several alternatives for the text that was recognized, in order of confidence. We'll only consider the result that is most likely to be correct, which is at index zero. After retrieving the result of the prompt, we display it using `CardBuilder`.

In `VoiceRecognitionActivity.java`:

```
public class VoiceRecognitionActivity extends Activity {
    private CardBuilder mCardBuilder;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mCardBuilder = new CardBuilder(this, CardBuilder.Layout.TEXT);

        ArrayList<String> voiceResults = getIntent().getExtras()
            .getStringArrayList(RecognizerIntent.EXTRA_RESULTS);
        String text = voiceResults.get(0);
        updateText(text);
    }

    private void updateText(String text) {
        mCardBuilder.setText(text);
        setContentView(mCardBuilder.getView());
    }
}
```

Standalone Voice Recognition

We can also start voice recognition directly from an activity by starting an intent with action `RecognizerIntent.ACTION_RECOGNIZE_SPEECH`, after which the results of voice recognition are passed into the `onActivityResult` method. In this example, we'll start voice recognition after the app has started and change the message that is displayed on the screen.

1. Declare the `SPEECH_REQUEST` constant, which is a request code used to start voice recognition.

In `VoiceRecognitionActivity.java`:

```
private static final int SPEECH_REQUEST = 1;
```

2. Implement `displaySpeechRecognizer`, which starts an intent that requests users to input a message with voice recognition.

The intent's `RecognizerIntent.EXTRA_PROMPT` extra specifies what message is displayed in the voice recognition prompt.

In `VoiceRecognitionActivity.java`:

```
private void displaySpeechRecognizer() {
    Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
    intent.putExtra(RecognizerIntent.EXTRA_PROMPT, "Say something");
    startActivityForResult(intent, SPEECH_REQUEST);
}
```

3. Receive the result of voice recognition in the `onActivityResult` method.

If `resultCode` is not equal to `RESULT_OK`, then the user dismissed the voice recognition prompt. If the `resultCode` is `RESULT_OK`, fetch the results of voice recognition from the `RecognizerIntent.EXTRA_RESULTS` extra. Similar to the previous example, the results of voice recognition contain several alternatives for the text that was recognized, in order of confidence. We'll only consider the result that is most likely to be correct, which is at index zero.

In `VoiceRecognitionActivity.java`:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
    Intent data) {
    if (requestCode == SPEECH_REQUEST && resultCode == RESULT_OK) {
        List<String> results = data.getStringArrayListExtra(
            RecognizerIntent.EXTRA_RESULTS);
        String spokenText = results.get(0);

        updateText(spokenText);
    }

    super.onActivityResult(requestCode, resultCode, data);
}
```

The only thing we're missing is a way to call `displaySpeechRecognizer`. We'll do it with contextual voice commands.

Contextual Voice Commands

Contextual voice commands are menus that can be opened and triggered with voice recognition. When contextual voice commands are available, Glass displays a small "Ok Glass" text, centered at the bottom of the screen (see Figure 10-8).

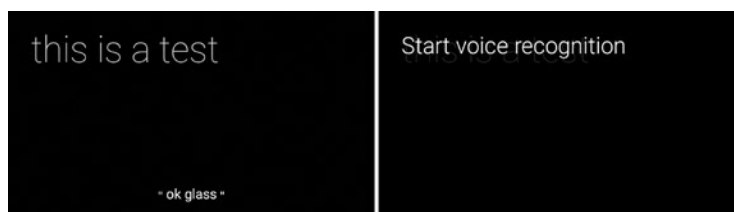


Figure 10-8. Using contextual voice commands. The "ok glass" text at the bottom of the screen indicates that contextual voice commands are available (top). Saying "ok glass" displays a list of menu items that can be selected with voice recognition (bottom)

This example demonstrates how to implement contextual voice commands.

1. Enable contextual voice commands.

Add the following code to the onCreate method.

In VoiceRecognitionActivity:

```
getWindow().requestFeature(WindowUtils.FEATURE_VOICE_COMMANDS);
```

2. Create a menu resource that contains all the desired menu items.

In res ► menu ► voicerec.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
<item
    android:id="@+id/voicerec"
    android:icon="@drawable/ic_launcher"
    android:title="Start voice recognition" />
</menu>
```

3. Override onCreatePanelMenu and onOptionsItemSelected.

By allowing the featureId of both of these methods to be either FEATURE_VOICE_COMMANDS or FEATURE_OPTIONS_PANEL, we allow this menu to be triggered with contextual voice commands or as a normal options menu. That is, these menus can be accessed with voice or with the touchpad. Note that there is no need to override onCreateOptionsMenu if we overrode onCreatePanelMenu.

In VoiceRecognitionActivity.java:

```
@Override
public boolean onCreatePanelMenu(int featureId, Menu menu) {
    if (featureId == WindowUtils.FEATURE_VOICE_COMMANDS ||
        featureId == Window.FEATURE_OPTIONS_PANEL) {
        getMenuInflater().inflate(R.menu.voicerec, menu);
        return true;
    }

    return super.onCreatePanelMenu(featureId, menu);
}

@Override
public boolean onOptionsItemSelected(int featureId, MenuItem item) {
    if (featureId == WindowUtils.FEATURE_VOICE_COMMANDS ||
        featureId == Window.FEATURE_OPTIONS_PANEL) {
        switch (item.getItemId()) {
            case R.id.voicerec:
                displaySpeechRecognizer();
                return true;
            default:
                return true;
        }
    }
}
```

```
    }  
  }  
  // Good practice to pass through to super if not handled  
  return super.onMenuItemSelected(featureId, item);  
}
```

4. Add the DEVELOPMENT to the manifest to enable using unlisted voice commands:

In `AndroidManifest.xml`:

```
<uses-permission android:name="com.google.android.glass.permission.DEVELOPMENT" />
```

Note Google maintains a set of approved contextual voice commands that can be used in production Glassware. An up-to-date list of these commands can be found in the documentation for `ContextualMenus.Command` (see <https://developers.google.com/glass/develop/gdk/reference/com/google/android/glass/app/ContextualMenus.Command>). Unlisted voice commands can be used only for development purposes with the `com.google.android.glass.permission.DEVELOPMENT` permission. Glassware that uses unlisted voice commands will not be approved for inclusion in the MyGlass store.

You have now finished creating the Glassware. Test it to see three different methods of invoking voice recognition.

Summary

We started this chapter by learning about the touchpad and the `MotionEvent`s it generates, and then we saw how to use `GestureDetector` to detect a variety of gestures (such as taps and swipes), determine how many fingers are on the touchpad, and calculate how much a user scrolls. Then, we implemented head gestures by leveraging the gyroscopic sensors of Glass. We concluded the chapter by learning about voice recognition and contextual voice commands. In the next chapter, we'll focus on the primary aspect that distinguishes Glass from other wearables such as Android Wear: the camera.

The Camera: Taking Pictures and Recording Video

The Glass camera provides a unique opportunity for innovation because, unlike ordinary cameras, Glass allows users to record experiences that occupy both hands such as riding a bicycle or skydiving. Additionally, Glass encourages users to take pictures while focusing on their experience instead of on the camera; if, for instance, users are recording a video of a concert, they can simply start recording and focus on the concert. The Glass camera records the concert without forcing users to stare at their viewfinders while holding their cameras up in the air. Starting a recording and letting it run with minimal intervention is particularly effective for time-lapses, as we'll see later this chapter.

Note A viewfinder is a term from photography and refers to the mechanism that allows the photographer to see where the camera is pointing. Optical viewfinders are eyepieces that contain lenses to show photographers a preview of the image. Digital cameras and mobile/wearable devices contain digital viewfinders, which show users a preview of the photograph or video on a digital screen. In Android development, the viewfinder is also known as the camera preview.

In addition to recording pictures or videos, Glass can leverage the camera to understand a user's environment (such as with object recognition) and to help users communicate (such as video chatting). The processing and networking capabilities of Glass enable capabilities that are beyond the scope of other cameras. While the specifications of the camera are ordinary—it can take photos at 5MP and videos at 720p—its potential for innovation is unlike any other camera.

In this chapter, we'll learn to take pictures and record videos on Glass in a variety of ways. In particular, we'll:

- leverage the native camera app to take pictures and record videos
- take pictures and record videos from an immersion directly with the camera API
- display a camera preview on a LiveCard
- record a time-lapse

Taking Pictures and Recording Videos with Intents

The native camera Glassware allows users to take pictures with the “Take a picture” voice command or by pressing the camera button at any time. Similarly, users can record videos with the “Record a video” voice command or by pressing and holding the camera button at any time. In either case, a new activity pops up, captures the media, and immediately closes. When users take a picture, in particular, they only see the image after it has been taken and do not see any sort of viewfinder or preview window.

Our Glassware can utilize the native camera app to capture media without having to deal with the camera API. This solution is ideal for most situations that do not require precisely framing pictures or processing individual frames of the camera preview.

Taking Pictures

Since the native camera Glassware takes care of all the heavy lifting, taking pictures is remarkably easy (except for one detail).

1. Start an Intent with action `MediaStore.ACTION_IMAGE_CAPTURE`:

```
Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
startActivityForResult(takePictureIntent, CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE);
```

2. If needed, utilize the captured image and its thumbnail in `onActivityResult`:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    ...
    String thumbnailPath = data.getStringExtra(Intent.EXTRA_THUMBNAIL_FILE_PATH);
    String picturePath = data.getStringExtra(Intent.EXTRA_PICTURE_FILE_PATH);
    // do something with the files in these directories
    processPictureWhenReady(picturePath);
    ...
}
```

When `onActivityResult` is called, the image's thumbnail is ready to be displayed, but the image itself may not have been completely written to memory. The native camera Glassware performs a significant amount of post-processing on the image that can take as long as 10 or 15 seconds.

If you just need to display the image, displaying its thumbnail is sufficient. However, you must wait for the image to be available if you need to use the full image to, for instance, share it over social media. Waiting for the image to be written requires using a `FileObserver`, as we'll see in a few sections.

The resolutions of the resulting image and its thumbnail are 2528x1856 and 624x352, respectively. These resolutions are hardcoded into the native camera app and cannot be modified.

Recording Videos

Recording videos with an `Intent` is similar to taking pictures.

1. Start an `Intent` with action `MediaStore.ACTION_IMAGE_CAPTURE`:

```
Intent recordVideoIntent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
startActivityForResult(recordVideoIntent, CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE);
```

When the recording is completed, `onActivityResult` will be called with the video's path.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    ...
    String thumbnailPath = data.getStringExtra(Intent.EXTRA_THUMBNAIL_FILE_PATH);
    String videoPath = data.getStringExtra(Intent.EXTRA_VIDEO_FILE_PATH);
    // do something with the files in these directories
    processPictureWhenReady(videoPath);
    ...
}
```

With `onActivityResult`, the video may not have been written to memory just as described in the previous section on taking pictures. The `processPictureWhenReady` method, which we'll implement in the next section, waits for this file to be available before attempting to use it.

Example #1: Capturing Media with Intents

This example displays a viewfinder and allows users to take a picture or record a video from the options menu. After taking a picture or recording a video, its thumbnail is displayed on the screen. When the native camera Glassware takes a picture, it can take 10 or 15 seconds for it to perform post processing and write the entire image to memory. In this example, we'll demonstrate how to wait for this image to be ready by:

1. displaying an indeterminate slider, and
2. disabling the ability to take new pictures until the image has been written to memory.

Define a Voice Command

Start the activity in response to the “Demo Camera Intent” voice command.

1. Create a voice trigger with the “Demo camera intent” keyword.

In res ► xml ► cameraintent_trigger.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Demo camera intent" />
```

2. Since this example uses an unlisted voice command, add the development permission.

In AndroidManifest.xml:

```
<uses-permission android:name="com.google.android.glass.permission.DEVELOPMENT" />
```

3. Configure the activity to be started in response to the voice command defined above.

```
<activity
  android:name=".CameraIntentActivity">
  <intent-filter>
    <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
  </intent-filter>

  <meta-data
    android:name="com.google.android.glass.VoiceTrigger"
    android:resource="@xml/cameraintent_trigger" />
</activity>
```

If you run the app at this stage, the “Demo camera intent” voice command should appear in the “OK Glass” menu, and it should start an empty activity.

Create a Layout

The activity’s layout contains an `ImageView` that displays the thumbnail of a captured photo or video and a view obtained with `CardBuilder` that displays the text *Tap to open the menu*. The former is added in XML and the latter is added programmatically in the next section.

1. Create an XML layout.

In res ► layout ► activity_cameraintent.xml:

```
<FrameLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/container"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
```

```

<ImageView
    android:id="@+id/image"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

</FrameLayout>

```

2. Declare constants and member variables.

In `CameraIntentActivity.java`:

```

public class CameraIntentActivity extends Activity {
    private static final int CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE = 1;
    private static final int CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE = 2;
    private ImageView mImage;
    private CardBuilder mCardBuilder;
    private AudioManager mAudioManager;
    private Slider.Indeterminate mIndeterminate;
    private FileObserver mObserver;
    ...
}

```

3. Implement the `onCreate` method.

The `onCreate` method takes the following actions:

- Adds the `FLAG_KEEP_SCREEN_ON` flag to prevent the screen of Glass from timing out and turning off.
- Wraps the main layout with a `TuggableView` to indicate that swiping on the main layout does not do anything.
- Adds a `CardBuilder` view that adds *Tap to open the menu* to the main layout of this activity.
- Sets the scale type of the `ImageView` that displays the thumbnail to `CENTER_CROP`.

In `CameraIntentActivity.java`:

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    super.onCreate(savedInstanceState);
    View content = getLayoutInflater().inflate(R.layout.activity_cameraintent, null);

    ViewGroup container = (ViewGroup) content.findViewById(R.id.container);
    mImage = (ImageView) content.findViewById(R.id.image);

    TuggableView tuggableView = new TuggableView(this, content);
    setContentView(tuggableView);
    mAudioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
}

```

```

mCardBuilder = new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText("Tap to open the menu");

container.addView(mCardBuilder.getView());

mImage.setScaleType(ImageView.ScaleType.CENTER_CROP);
mSlider = Slider.from(tuggableView);
}

```

The scale type of the `ImageView` is set to center crop to ensure that the image fills the entire screen even though a thumbnail is smaller than the screen. With XE22, the picture's thumbnail is 624x352 and the video's thumbnail is 512x288, in contrast to the screen size of 640x360. A scale type of center crop increases the size of the image while maintaining its aspect ratio and ensures that the image covers the entire screen.

Note `TuggableView` encapsulates a list of cards (that is, a `CardScrollView`) that only has a single item, which is given as a parameter in its constructor. `TuggableView` uses `CardScrollView` to provide tugging feedback to indicate that swiping has no effect. Note that the implementation of `TuggableView`, which was thoroughly explained in Chapter 8, is not explained in this chapter.

Create an Options Menu

The options menu only contains two menu items: *Take Picture* and *Record Video*.

1. Create the resource for the menu.

In res ► menu ► camera_intent.xml:

```

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/action_take_picture"
    android:icon="@drawable/ic_camera_50"
    android:title="Take Picture" />
  <item
    android:id="@+id/action_record_video"
    android:icon="@drawable/ic_video_50"
    android:title="Record Video" />
</menu>

```

The `ic_camera_50` and the `ic_video_50` icons are available in the book's example source code and can also be downloaded from the official documentation (see <https://developers.google.com/glass/tools-downloads/downloads>).

2. In `onCreateOptionsMenu`, create a menu with the resource from the previous step.

In `CameraIntentActivity.java`:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    getMenuInflater().inflate(R.menu.camera_intent, menu);
    return true;
}
```

3. Take an appropriate action when each menu item is selected.

If `mCapturingMedia` is true, then the Glassware is currently processing a previous picture. In this case, play the `DISALLOWED` sound to notify a user that the action is not yet available.

In `CameraIntentActivity.java`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if(mCapturingMedia) {
        mAudioManager.playSoundEffect(Sounds.DISALLOWED);
        return true;
    }

    switch(item.getItemId()) {
        case R.id.action_take_picture:
            startTakePictureIntent();
            return true;
        case R.id.action_record_video:
            startRecordVideoIntent();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

The `startTakePictureIntent` and `startRecordVideoIntent` methods are implemented below.

4. Open the options menu in response to a tap.

When a user taps on the touchpad, play the `TAP` sound in addition to opening the options menu. Recall that Glass automatically translates certain gestures into D-pad key events. Namely, tapping the touchpad triggers a key event with a key code of `KEYCODE_DPAD_CENTER`.

In `CameraIntentActivity.java`:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
        AudioManager.playSoundEffect(Sounds.TAP);
        openOptionsMenu();
        return true;
    }
    return super.onKeyDown(keyCode, event);
}
```

Capture the Media and Handle `onActivityResult`

The most important part of this activity issues the commands to capture the media and receives the result.

1. Implement the `startTakePictureIntent` and `startRecordVideoIntent` methods.

These methods start the appropriate intent and set `mCapturingMedia` to true to indicate that the activity is currently processing an existing picture or video. Users cannot capture media until the previous picture has been written to memory.

In `CameraIntentActivity.java`:

```
private void startTakePictureIntent() {
    Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    startActivityForResult(takePictureIntent, CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE);
    mCapturingMedia = true;
}

private void startRecordVideoIntent() {
    Intent recordVideoIntent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
    startActivityForResult(recordVideoIntent, CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE);
    mCapturingMedia = true;
}
```

2. In the `onActivityResult` method, display the thumbnail of the picture or video that was just captured.

If a picture was taken, start an indeterminate slider and call the `processPictureWhenReady`, which waits until the picture has been written to memory before hiding the slider and setting `mCapturingMedia` to false.

In `CameraIntentActivity.java`:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if(resultCode == RESULT_CANCELED) {
        // user cancelled the image or video capture
        mCapturingMedia = false;
        return;
    }
}
```

```

    } else if(resultCode != RESULT_OK) {
        // capture failed, advise user
        Toast.makeText(this, "Error capturing media", Toast.LENGTH_SHORT).show();
        mCapturingMedia = false;
        return;
    }

    if (requestCode == CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE) {
        final String picturePath = data.getStringExtra(Intent.EXTRA_PICTURE_FILE_PATH);
        // do something with picturePath if needed
        mIndeterminate = mSlider.startIndeterminate();
        processPictureWhenReady(picturePath);
    } else if(requestCode == CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE) {
        String videoPath = data.getStringExtra(Intent.EXTRA_VIDEO_FILE_PATH);
        // do something with videoPath if needed
        mCapturingMedia = false;
    }

    String thumbnailPath = data.getStringExtra(Intent.EXTRA_THUMBNAIL_FILE_PATH);
    displayImage(thumbnailPath);

    super.onActivityResult(requestCode, resultCode, data);
}

```

After users capture a picture or a video, the native camera shows them its thumbnail and allows them to accept the image by tapping or reject it by swiping down. If the user does not accept the image, `onActivityResult` will be called with a result code of `RESULT_CANCELED`. In this case, no further action is necessary.

1. Implement the `processPictureWhenReady` method.

This method was derived from the official documentation (see <https://developers.google.com/glass/develop/gdk/camera>). If the image exists, this method hides the indeterminate slider, sets `mCapturingMedia` to false, and displays a Toast message.

If the image has not yet been written to memory, this method creates a `FileObserver` that watches for changes within the image's parent folder. When the image is written to memory, the `FileObserver` calls the `onEvent` callback.

In the `onEvent` callback, this method makes sure that the event corresponds to the image file and not any other file. If the `FileObserver` event corresponds to the image file, this method calls `processPictureWhenReady` recursively to process the image.

```

private void processPictureWhenReady(final String picturePath) {
    final File pictureFile = new File(picturePath);

    if (pictureFile.exists()) {
        Toast.makeText(getApplicationContext(), "picture is now ready for use",
            Toast.LENGTH_SHORT).show();
        mCapturingMedia = false;
        mIndeterminate.hide();
        mIndeterminate = null;
    }
}

```



```

} else {
    final File parentDirectory = pictureFile.getParentFile();
    mObserver = new FileObserver(parentDirectory.getPath(),
        FileObserver.CLOSE_WRITE | FileObserver.MOVED_TO) {
        // Protect against additional pending events after CLOSE_WRITE
        // or MOVED_TO is handled.
        private boolean isFileWritten;

        @Override
        public void onEvent(int event, String path) {
            if (!isFileWritten) {
                // For safety, make sure that the file that was created in
                // the directory is actually the one that we're expecting.
                File affectedFile = new File(parentDirectory, path);
                isFileWritten = affectedFile.equals(pictureFile);

                if (isFileWritten) {
                    stopWatching();

                    // Now that the file is ready, recursively call
                    // processPictureWhenReady again (on the UI thread).
                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            processPictureWhenReady(picturePath);
                        }
                    });
                }
            }
        }
    };

    mObserver.startWatching();
}

```

Note that the `FileObserver` object called `mObserver` as a member variable. In the official documentation, on the other hand, this object is a local variable. When running this example on XE22 with `mObserver` as a local variable, the `FileObserver` does not always call the `onEvent` callback. I hypothesize that the local `FileObserver` object can get garbage collected before being able to trigger the callback method. Making `mObserver` into a member variable protects it from garbage collection. Since making that change, `FileObserver` always triggers the `onEvent` callback on my Glass.

2. When the image is available, load it in a background task and then display it in the `ImageView`.

Loading an image is not necessarily fast, and doing so in a background thread is good practice. Since the thumbnail of a picture or video is smaller than the screen, there is no need to resize the image. The `ImageView`'s `CENTER_CROP` scale type takes care of increasing the image size to fill the entire screen.

```

private class LoadAndSetImageTask extends AsyncTask<String, Void, Bitmap> {
    private WeakReference<ImageView> mImageViewWeakReference;

    public LoadAndSetImageTask(ImageView imageView) {
        mImageViewWeakReference = new WeakReference<ImageView>(imageView);
    }

    @Override
    protected Bitmap doInBackground(String... params) {
        return BitmapFactory.decodeFile(params[0]);
    }

    @Override
    protected void onPostExecute(Bitmap bitmap) {
        super.onPostExecute(bitmap);

        if (mImageViewWeakReference != null && bitmap != null) {
            final ImageView imageView = mImageViewWeakReference.get();
            if (imageView != null) {
                imageView.setImageBitmap(bitmap);
            }
        }
    }
}

```

3. Override `onOptionsMenuClosed` to re-display the indeterminate slider if needed.

Recall that a slider is a global component. If a user opens the options menu while viewing an indeterminate slider, the options menu disables the indeterminate slider when creating a scroller slider. When the options menu is closed, the indeterminate slider disappears. This issue is addressed in the `onOptionsMenuClosed` method, which restarts the indeterminate slider when the options menu is closed while `mCapturingMedia` is still true.

```

@Override
public void onOptionsMenuClosed(Menu menu) {
    super.onOptionsMenuClosed(menu);
    if (mIndeterminate != null && mCapturingMedia) {
        mIndeterminate.show();
    }
}

```

The next example will take pictures without relying on the native camera app.

Example #2: Using the Camera API

The Camera API allows Glassware to access the camera directly and to display a viewfinder, which is also referred to as a camera preview. The Camera API renders the camera preview on a `SurfaceView`, which is a view that can be updated from a background thread.

This example demonstrates how to use the Camera API to display a viewfinder and take pictures. Parts of the following code were derived from the official documentation (see <http://developer.android.com/guide/topics/media/camera.html>).

Implementing CameraUtils

In CameraUtils, we write a few helper methods that are useful for Glassware that uses the Camera API.

1. Implement `getCameraInstance`, which attempts to open and return a Camera object.

If the camera is in use or unavailable, this method returns null.

In CameraUtils.java:

```
public static Camera getCameraInstance() {
    Camera c = null;
    try {
        c = Camera.open(); // attempt to get a Camera instance
    }
    catch (Exception e){
        // Camera is not available (in use or does not exist)
    }
    return c; // returns null if camera is unavailable
}
```

2. Implement `getOutputMediaFile`, which generates a File with a filename that contains a prefix, timestamp, and extension.

When plugged in to a computer, Glass uses the Picture Transfer Protocol (PTP) to appear as a digital camera. Users can then access the DCIM and the Pictures directories of Glass from the computer. In this example, we place all pictures into the `AndroidWearables` folder within the Pictures directory (that is, in `/sdcard/Pictures/AndroidWearables`).

The filenames generated by this method have the following format: PREFIX_TIMESTAMP.EXTENSION where `TIMESTAMP` is of the format `yyyyMMdd_HHmms`. For instance, the filename with the prefix `IMG` and the extension `jpg` that was created on March 3, 2014 at 11:15:02am would be `IMG_20140303_111502.jpg`.

In CameraUtils.java:

```
private static File getOutputMediaFile(String prefix, String extension){
    // To be safe, you should check that the SDCard is mounted
    // using Environment.getExternalStorageState() before doing this.

    File mediaStorageDir = new File(Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_PICTURES), "AndroidWearables");
    // This location works best if you want the created images to be shared
    // between applications and persist after your app has been uninstalled.
```

```

// as opposed to getExternalFilesDir

// Create the storage directory if it does not exist
if (! mediaStorageDir.exists()){
    if (! mediaStorageDir.mkdirs()){
        Log.d("AndroidWearables", "failed to create directory");
        return null;
    }
}

// Create a media file name
String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
File mediaFile;
mediaFile = new File(mediaStorageDir.getPath() + File.separator +
    prefix + "_" + timeStamp + "." + extension);

return mediaFile;
}

```

3. Implement `getOutputPictureFile`, which uses the method implemented in the previous step to create a filename with a prefix of `IMG` and an extension of `jpg`.

In `CameraUtils.java`:

```

public static File getOutputPictureFile() {
    return getOutputMediaFile("IMG", "jpg");
}

```

4. Implement the `scanFile` method, which notifies the media scanner that a new file has been added.

Consider the case in which Glassware creates a new file in the Pictures directory. Normally, a recently created file is not visible from a USB connected computer unless Glass has been restarted. Calling the `scanFile` method sends a broadcast that tells the media scanner to take into account a new file. After calling the `scanFile` method, the file should be visible from the computer without having to restart Glass.

In `CameraUtils.java`:

```

public static void scanFile(Context context, String filename) {
    context(getApplicationContext()).sendBroadcast(
        new Intent("android.intent.action.MEDIA_SCANNER_SCAN_FILE",
            Uri.fromFile(new File(filename))));
}

```

The methods in `CameraUtils` are used throughout the rest of this example.

Implementing the SurfaceHolder Callback

The `CameraPreview` class, which extends `SurfaceView`, is the view on which the camera preview is rendered. A `SurfaceView` should not be modified directly, but through its `SurfaceHolder`. `SurfaceHolder` provides an interface to draw on the pixels of a `SurfaceView` and to monitor its dimensions and status.

`SurfaceHolder` contains a callback (that is, `SurfaceHolder.Callback`) with the following methods:

- **surfaceCreated:** Indicates that the `SurfaceView` has been initialized and can be used.
- **surfaceChanged:** This method is called any time the dimensions of the `SurfaceView` change. If the dimensions of the `SurfaceView` never change, the `surfaceChanged` method only gets called once after `surfaceCreated`.
- **surfaceDestroyed:** Indicates that the `SurfaceView` should no longer be utilized.

This example initializes the camera parameters and starts the camera preview within the `surfaceChanged` callback.

A camera's parameters define several settings that affect the way the camera operates. With Glass, we must set three parameters:

- **preview FPS range:** Defines a range of values for the frame rate of the camera preview. With Glass, a range of (30000, 30000) is most reliable.
- **preview size:** Defines the dimensions of the camera preview. With Glass, a preview size of 640x360 occupies the entire screen.
- **picture size:** Defines the dimensions of the pictures that the camera takes. We'll use the same dimensions used by the native camera Glassware, which is 2528x1856.

In addition to setting the parameters, we must:

1. tell the camera where to draw the camera preview (by calling `setPreviewDisplay`), and
2. tell the camera to start rendering the camera preview on the preview display (by calling `startPreview`).

In `CameraPreview.java`:

```
public class CameraPreview extends SurfaceView implements SurfaceHolder.Callback {
    private static final String TAG = "CAM";
    private SurfaceHolder mHolder;
    private Camera mCamera;
```


Implementing CameraActivity

The activity's layout contains an `ImageView`, which displays the thumbnail of a captured photo or video, and a view obtained with `CardBuilder` that displays the text *Tap to open the menu*. The options menu contains a single item called *Take a picture* that uses the Camera API to take a picture.

1. Create a voice trigger with the "Demo camera API" keyword.

In res ► xml ► camera_trigger.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Demo camera API" />
```

2. Since we're using an unlisted voice command, add the development permission.

In `AndroidManifest.xml`:

```
<uses-permission android:name="com.google.android.glass.permission.DEVELOPMENT" />
```

3. Configure the activity to be started in response to the voice command defined in step 1.

In `AndroidManifest.xml`:

```
<activity
    android:name=".CameraActivity">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>

    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/camera_trigger" />
</activity>
```

4. Create a layout.

This layout contains an empty `FrameLayout`.

In res ► layout ► activity_camera.xml:

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

</FrameLayout>
```

Initialize the Layout and SoundPool

Declare member variables, instantiate the layout, and initialize the `SoundPool` in the `onCreate` method.

1. Declare constants and member variables.

In `CameraActivity.java`:

```
public class CameraActivity extends Activity {
    private static final String TAG = "CA";
    private Camera mCamera;
    private CameraPreview mPreview;
    private TuggableView mTuggableView;
    private volatile boolean mTakingPicture;
    private SoundPool mSoundPool;
    private float mVolume;
    private AudioManager mAudioManager;
    private int mPhotoReadySoundId, mPhotoShutterSoundId;
    ...
}
```

2. Override the `onCreate` method.

This method:

- inflates the layout and wraps it in a `TuggableView` to provide tuggable feedback.
- initializes a camera and a camera preview.
- adds the camera preview to the layout.
- adds an empty `TextView` to the layout, on top of the camera preview (we'll see why shortly).
- initializes two sound files: the photo ready sound, which is played as soon as a user taps on the *Take picture* menu item, and the photo shutter sound, which is played right after the picture has been taken.

In `CameraActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    super.onCreate(savedInstanceState);
    View content = getLayoutInflater().inflate(R.layout.activity_camera, null);
    mTuggableView = new TuggableView(this, content);
    setContentView(mTuggableView);

    mCamera = CameraUtils.getCameraInstance();

    mPreview = new CameraPreview(this, mCamera, false);
    FrameLayout preview = (FrameLayout) content.findViewById(R.id.container);
    preview.addView(mPreview);
    preview.addView(new TextView(this));
}
```



```

mAudioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);

mSoundPool = new SoundPool(2, AudioManager.STREAM_MUSIC, 0);
mPhotoReadySoundId = mSoundPool.load(getApplicationContext(),
    R.raw.sound_photo_ready, 1);
mPhotoShutterSoundId = mSoundPool.load(getApplicationContext(),
    R.raw.sound_photo_shutter, 1);

float actVolume = (float) mAudioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
float maxVolume = (float) mAudioManager.getStreamMaxVolume(AudioManager.STREAM_MUSIC);
mVolume = actVolume / maxVolume;
}

```

The empty `TextView` we add to the layout may seem useless, but it serves a purpose. At least on XE22, the camera preview has a bug where it swaps an image's red and blue channels. This bug disrupts the appearance of the camera preview and, for instance, makes people's skin look blue. To avoid this bug, add an empty view at the end of the layout. Note that future versions of Glass may fix this problem, but for now this workaround is necessary.

3. Create a resource for a menu with a single item called *Take Picture*.

In `res > menu > camera.xml`:

```

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_take_picture"
        android:icon="@drawable/ic_camera_50"
        android:title="Take Picture" />
</menu>

```

4. In `onCreateOptionsMenu`, create a menu with the resource from the previous step.

In `CameraActivity.java`:

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    getMenuInflater().inflate(R.menu.camera, menu);
    return true;
}

```

5. Create the options menu in the activity and handle its item click.

When a user taps on the *Take a picture* menu item, use the Camera API to take a picture.

In `CameraActivity.java`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        case R.id.action_take_picture:
            takePicture();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

6. When taking a picture, play a sound to give the user feedback.

In the rare case in which a user tries to take a picture while the previous picture is still processing, play the `DISALLOWED` sound. This case can only occur if a user tries to take multiple pictures in rapid succession.

To take a picture, call the `Camera` object's `takePicture` method, which takes a `PictureCallback` as its third parameter. A `PictureCallback` gets notified when the picture has been taken and is available as a jpeg byte array.

In `CameraActivity.java`:

```
private void takePicture() {
    if (mTakingPicture) {
        AudioManager.playSoundEffect(Sounds.DISALLOWED);
    } else {
        mTakingPicture = true;
        mCamera.takePicture(null, null, mPicture);
        playSound(mPhotoReadySoundId);
    }
}
```

7. The `playSound` method simply wraps `SoundPool` for convenience.

Since the `Camera` API does not provide any feedback when taking a picture, we'll use a `SoundPool` to play an audio file when taking a picture.

In `CameraActivity.java`:

```
private void playSound(int soundId) {
    mSoundPool.play(soundId, mVolume, mVolume, 1, 0, 1f);
}
```

8. Open the options menu when a user taps on the touchpad. Additionally, take a picture when a user presses the camera button.

Recall that tapping on the touchpad generates a key down event with key code `KEYCODE_DPAD_CENTER`. Additionally, pressing the camera button generates a key down event with key code `KEYCODE_CAMERA`. When the user taps on the touchpad, we must play the TAP sound and open the options menu. When a user presses the camera button, we must take a picture.

In `CameraActivity.java`:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
        AudioManager.playSoundEffect(Sounds.TAP);
        openOptionsMenu();
        return true;
    } else if (keyCode == KeyEvent.KEYCODE_CAMERA) {
        takePicture();
        return true;
    } else {
        return super.onKeyDown(keyCode, event);
    }
}
```

9. Release the camera if the user leaves the activity (that is, in `onPause`) and re-acquire the camera if the user returns (that is, in `onResume`).

In `CameraActivity.java`:

```
@Override
protected void onResume() {
    super.onResume();

    if (mCamera == null) {
        mCamera = CameraUtils.getCameraInstance();
    }
}

@Override
protected void onPause() {
    if (mCamera != null){
        mCamera.stopPreview();
        mCamera.release(); // release the camera for other applications
        mCamera = null;
    }

    super.onPause();
}
```

10. Implement `PictureCallback` to process the image once it has been taken.

The image is saved into a file in another thread to ensure that the UI thread is not occupied by the file writing operation, which can be slow. The image is saved into a file that is obtained with `CameraUtils.getOutputPictureFile`, which specifies a name with a timestamp and a jpg extension. After the image is saved into the file, we call `CameraUtils.scanFile` to notify the media scanner of the new file.

In `CameraActivity.java`:

```
private Camera.PictureCallback mPicture = new Camera.PictureCallback() {

    @Override
    public void onPictureTaken(final byte[] data, Camera camera) {
        playSound(mPhotoShutterSoundId);

        new Thread(new Runnable() {
            @Override
            public void run() {
                File pictureFile = CameraUtils.getOutputPictureFile();
                if (pictureFile == null){
                    Log.d(TAG, "Error creating media file, check storage permissions");
                    return;
                }

                try {
                    FileOutputStream fos = new FileOutputStream(pictureFile);
                    fos.write(data);
                    fos.close();

                    CameraUtils.scanFile(CameraActivity.this, pictureFile.
                        getAbsolutePath());
                } catch (FileNotFoundException e) {
                    Log.d(TAG, "File not found: " + e.getMessage());
                } catch (IOException e) {
                    Log.d(TAG, "Error accessing file: " + e.getMessage());
                }

                mTakingPicture = false;
            }
        }).start();

        mCamera.startPreview();
    }
};
```

At this point, you can run the example. Try taking pictures both by pressing the camera button and with the *Take Picture* menu item.

Example #3: Using the Camera API from a LiveCard

The camera preview is not restricted to running on activities and can also run in LiveCards, as demonstrated by this example.

Create LiveCameraPreview

In the previous example, we created a class called `CameraPreview` that extended `SurfaceView` and implemented `SurfaceCallbacks`. In this example, we'll create a similar class that is used with a `LiveCard`. `LiveCameraPreview` will not extend `SurfaceView` since the `SurfaceView` is contained within the `LiveCard`. This class will just implement `DirectRenderingCallback`, which is a sub-interface of `SurfaceCallbacks`. It has the same methods with the addition of `renderingPaused`, which notifies the callback when a user scrolls to and away from the `LiveCard`. Rendering the `LiveCard` when it's not displayed on the screen would be a waste of resources, and the `renderingPaused` method lets us pause or resume rendering when appropriate.

1. Declare constants and member variables.

In `LiveCameraPreview.java`:

```
public class LiveCameraPreview implements DirectRenderingCallback {
    private static final String TAG = "CAM";
    private SurfaceHolder mHolder;
    private Camera mCamera;
    private boolean mPreviewEnabled;
    ...
}
```

2. In the constructor, obtain the `SurfaceHolder` directly from the `LiveCard`.

In `LiveCameraPreview.java`:

```
public LiveCameraPreview(LiveCard liveCard, Camera camera) {
    mCamera = camera;

    // Install a SurfaceHolder.Callback so we get notified when the
    // underlying surface is created and destroyed.
    mHolder = liveCard.getSurfaceHolder();
    mHolder.addCallback(this);
}
```

3. Implement the `surfaceCreated` method, which is a part of `DirectRenderingCallback`.

This method has no content because we initialize the camera in the `surfaceChanged` method.

In `LiveCameraPreview.java`:

```
@Override
public void surfaceCreated(SurfaceHolder holder) {
}
```

4. Implement the `surfaceDestroyed` method.

This method indicates that the `LiveCard` was removed. Here, stop the camera preview and call the camera's `release` method to free its resources.

In `LiveCameraPreview.java`:

```
@Override
public void surfaceDestroyed(SurfaceHolder holder) {
    mCamera.stopPreview();
    mCamera.release();
}
```

5. Implement the `surfaceChanged` method.

This method is nearly identical to the implementation in `CameraPreview` from the previous example, with the exception that it sets `mPreviewEnabled` to `true` after initializing the camera. The `mPreviewEnabled` boolean indicates whether the camera preview is started.

In `LiveCameraPreview.java`:

```
@Override
public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
    if (mHolder.getSurface() == null){
        // preview surface does not exist
        return;
    }

    // stop preview before making changes
    try {
        mCamera.stopPreview();
    } catch (Exception e){
        // ignore: tried to stop a non-existent preview
    }

    // set preview size and make any resize, rotate or
    // reformatting changes here
    Camera.Parameters parameters = mCamera.getParameters();
    parameters.setPreviewFpsRange(30000, 30000);
    parameters.setPreviewSize(640, 360);
    parameters.setPictureSize(2528, 1856);
    parameters.setRecordingHint(true);
    mCamera.setParameters(parameters);
}
```

```

// start preview with new settings
try {
    mCamera.setPreviewDisplay(mHolder);
    mCamera.startPreview();
    mPreviewEnabled = true;
} catch (Exception e){
    Log.d(TAG, "Error starting camera preview: " + e.getMessage());
}
}

```

6. Override the `renderingPaused` method.

This method stops the camera preview when the `LiveCard` is no longer visible on the screen and starts the camera preview when the `LiveCard` once again enters the screen.

In `LiveCameraPreview.java`:

```

@Override
public void renderingPaused(SurfaceHolder holder, boolean pause) {
    if (pause && mPreviewEnabled) {
        mCamera.stopPreview();
        mPreviewEnabled = false;
    } else if(!pause && !mPreviewEnabled) {
        mCamera.startPreview();
        mPreviewEnabled = true;
    }
}
}

```

Creating a Menu for the `LiveCard`

This `LiveCard`'s menu only has a single menu item called *Stop* that removes the `LiveCard` from the timeline. Recall from chapter 9 that the menu of a `LiveCard` is implemented with an activity.

1. Create an XML resource for the menu.

In `res > menu > livecamera.xml`:

```

<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_stop"
        android:icon="@drawable/ic_video_50"
        android:title="Stop" />
</menu>

```

2. Create a style that gives the activity a translucent background.

In `res > values > styles.xml`:

```

<style name="MenuTheme" parent="@android:style/Theme.DeviceDefault">
    <item name="android:windowBackground">@android:color/transparent</item>
    <item name="android:colorBackgroundCacheHint">@null</item>
    <item name="android:windowIsTranslucent">true</item>
    <item name="android:windowAnimationStyle">@null</item>
</style>

```

3. Declare `CameraMenuActivity` in the manifest and set its theme to `MenuTheme`.

In `AndroidManifest.xml`:

```
<activity android:name=".CameraMenuActivity"
    android:theme="@style/MenuTheme"/>
```

4. Implement `CameraMenuActivity`.

This activity displays an options menu as soon as it starts. The menu only contains an item called *Stop* that removes the `LiveCard` from the timeline.

In `CameraMenuActivity.java`:

```
public class CameraMenuActivity extends Activity {
    @Override
    public void onAttachedToWindow() {
        super.onAttachedToWindow();
        openOptionsMenu();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);
        getMenuInflater().inflate(R.menu.livecamera, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch(item.getItemId()) {
            case R.id.action_stop:
                stopService(new Intent(this, CameraLiveCardService.class));
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }

    @Override
    public void onOptionsItemSelected(MenuItem item) {
        finish();
    }
}
```

Now that we created a menu, we can implement the `LiveCard`.

Implement CameraLiveCardService

This service is just like any other service that creates a LiveCard, with the exception that its renderer is LiveCameraPreview.

1. Create a voice trigger with the “Start live camera” keyword.

In res ► xml ► livecamera_trigger.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Start live camera" />
```

2. Since we’re using an unlisted voice command, add the development permission.

In AndroidManifest.xml:

```
<uses-permission android:name="com.google.android.glass.permission.DEVELOPMENT" />
```

3. Configure the service to be started in response to the voice command defined above.

In AndroidManifest.xml:

```
<service android:name=".CameraLiveCardService">
  <intent-filter>
    <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
  </intent-filter>

  <meta-data
    android:name="com.google.android.glass.VoiceTrigger"
    android:resource="@xml/livecamera_trigger" />
</service>
```

4. Implement CameraLiveCardService.

In CameraLiveCardService.java:

```
public class CameraLiveCardService extends Service {
  private static final String LIVECARD_TAG = "CamCard";
  private LiveCard mLiveCard;
  private Camera mCamera;

  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    publishCard();
    return START_NOT_STICKY;
  }
}
```

```

private void publishCard() {
    if(mLiveCard == null) {
        mLiveCard = new LiveCard(this, LIVECARD_TAG);

        Intent menuIntent = new Intent(this, CameraMenuActivity.class);
        menuIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK|Intent.FLAG_ACTIVITY_CLEAR_TASK);
        mLiveCard.setAction(PendingIntent.getActivity(this, 0, menuIntent, 0));

        mLiveCard.setDirectRenderingEnabled(true);

        mCamera = Camera.open();
        LiveCameraPreview renderer = new LiveCameraPreview(mLiveCard, mCamera);
        mLiveCard.publish(LiveCard.PublishMode.REVEAL);
    }
}

@Override
public void onDestroy() {
    unpublishCard();
    super.onDestroy();
}

private void unpublishCard() {
    if(mLiveCard != null) {
        mLiveCard.unpublish();
        mLiveCard = null;
    }
}

@Override
public IBinder onBind(Intent intent) {
    return null;
}
}

```

Run the example and verify that you can see the camera preview inside a LiveCard.

Example #4: Recording a Time-lapse Video

The Camera API can record a time-lapse: a video in which the frame rate is extremely low. A time-lapse summarizes a long event into a short video. For instance, recording a time-lapse on Glass while riding a bike would create a video that highlights the entire route in a short amount of time. This example records one frame every 2 seconds. Playing the resulting movie at 30 fps means that a minute of real time corresponds to a second of video.

Implementing TimelapseActivity

This activity displays the camera preview on the screen and allows users to start or stop recording a time-lapse by either pressing the camera button or by tapping on a menu item.

1. Create a voice trigger with the “Start time-lapse” keyword.

In res ► xml ► timelapse_trigger.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Start timelapse" />
```

2. Since we’re using an unlisted voice command, add the development permission.

In AndroidManifest.xml:

```
<uses-permission android:name="com.google.android.glass.permission.DEVELOPMENT" />
```

3. Configure the activity to be started in response to the voice command defined above.

In AndroidManifest.java:

```
<activity android:name=".TimelapseActivity">
  <intent-filter>
    <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
  </intent-filter>

  <meta-data
    android:name="com.google.android.glass.VoiceTrigger"
    android:resource="@xml/timelapse_trigger" />
</activity>
```

4. Create a layout.

This example uses the activity_camera.xml layout from example 2, which only contains a `FrameLayout` with id `"@+id/container"`.

5. Declare constants and member variables.

In `TimelapseActivity.java`:

```
public class TimelapseActivity extends Activity {
  private static final String TAG = "CA";
  private Camera mCamera;
  private CameraPreview mPreview;
  private TuggableView mTuggableView;
  private MediaRecorder mMediaRecorder;
  private boolean mIsRecording;
  private SoundPool mSoundPool;
  private int mStartRecordingSoundId, mStopRecordingSoundId;
```

```
private float mVolume;
private AudioManager mAudioManager;
private String mOutputFilename;
...
```

6. Override the onCreate method.

This method is similar to the previous example's onCreate and

- inflates the layout and wraps it in a TuggableView to provide tuggable feedback.
- initializes a camera and a camera preview.
- adds the camera preview to the layout.
- adds an empty TextView to the layout, on top of the camera preview (we'll see why shortly).
- initializes two sound files.

In `TimelapseActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    super.onCreate(savedInstanceState);
    View content = getLayoutInflater().inflate(R.layout.activity_camera, null);
    mTuggableView = new TuggableView(this, content);
    setContentView(mTuggableView);

    mCamera = CameraUtils.getCameraInstance();

    mPreview = new CameraPreview(this, mCamera, true);
    FrameLayout preview = (FrameLayout) content.findViewById(R.id.container);
    preview.addView(mPreview);
    preview.addView(new TextView(this)); // needed to avoid bug

    initSoundPool();
}
```

Recall that adding the empty TextView to the end of the layout prevents a bug that causes the camera preview to display with its red and blue channels swapped.

7. In the `initSoundPool` method, initialize two sounds that play when a user starts or stops recording a time-lapse, respectively.

In `TimelapseActivity.java`:

```
private void initSoundPool() {
    mAudioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);

    mSoundPool = new SoundPool(2, AudioManager.STREAM_MUSIC, 0);
    mStartRecordingSoundId = mSoundPool.load(getApplicationContext(), R.raw.sound_video_start, 1);
    mStopRecordingSoundId = mSoundPool.load(getApplicationContext(), R.raw.sound_video_stop, 1);
}
```

```
float actVolume = (float) mAudioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
float maxVolume = (float) mAudioManager.getStreamMaxVolume(AudioManager.STREAM_MUSIC);
mVolume = actVolume / maxVolume;
}
```

8. The `playSound` method simply wraps `SoundPool` for convenience.

In `TimelapseActivity.java`:

```
private void playSound(int soundId) {
    mSoundPool.play(soundId, mVolume, mVolume, 1, 0, 1f);
}
```

9. Release the camera in `onPause` to ensure the camera is never running in the background. Additionally, reacquire the camera in `onResume` to ensure the camera's preview is enabled any time the activity is in the foreground.

In `TimelapseActivity.java`:

```
@Override
protected void onResume() {
    super.onResume();

    if (mCamera == null) {
        mCamera = getCameraInstance();
    }
}

@Override
protected void onPause() {
    if (mCamera != null) {
        mCamera.stopPreview();
        mCamera.release(); // release the camera for other applications
        mCamera = null;
    }

    super.onPause();
}
```

Creating an Options Menu

When a user is recording a time-lapse, the options menu contains a single item titled *Stop time-lapse*. When a user is not recording a time-lapse, the menu item is titled *Record time-lapse*.

1. Create a resource for the menu.;

In res ► menu ► timelapse.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/action_record_timelapse"
    android:icon="@drawable/ic_video_50"
    android:title="Record timelapse" />
</menu>
```

2. In onCreateOptionsMenu, create a menu with the resource from the previous step.

In TimelapseActivity.java:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    getMenuInflater().inflate(R.menu.timelapse, menu);
    return true;
}
```

3. Change the menu item's icon and label depending on whether the time-lapse is currently recording or not.

When recording a time-lapse, the menu should display a single item, *Stop time-lapse* and the `ic_video_off_50` icon. When not recording a time-lapse, the menu should display a single item, *Record time-lapse* and the `ic_video_50` icon.

In TimelapseActivity.java:

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    super.onPrepareOptionsMenu(menu);
    if(mIsRecording) {
        menu.findItem(R.id.action_record_timelapse).setIcon(R.drawable.ic_video_off_50);
        menu.findItem(R.id.action_record_timelapse).setTitle("Stop timelapse");
    } else {
        menu.findItem(R.id.action_record_timelapse).setIcon(R.drawable.ic_video_50);
        menu.findItem(R.id.action_record_timelapse).setTitle("Record timelapse");
    }
    return true;
}
```

The `ic_video_50` and `ic_video_off_50` icons are available in the book's example source code and can also be downloaded from the official documentation (see <https://developers.google.com/glass/tools-downloads/downloads>).

4. When the menu item is selected, toggle the state of the time-lapse recording. That is, stop the recording if it's running and start it if the recording is stopped.

Additionally, call `invalidateOptionsMenu` after toggling the state of the recording to notify the system that the menu should be updated by calling `onPrepareOptionsMenu`.

In `TimelapseActivity.java`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_record_timelapse:
            toggleRecording();
            invalidateOptionsMenu();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

The `toggleRecording` method is implemented shortly.

5. Open the options menu when a user taps on the touchpad. Additionally, start or stop recording the time-lapse when a user presses the camera button.

In `TimelapseActivity.java`:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
        AudioManager.playSoundEffect(Sounds.TAP);
        openOptionsMenu();
        return true;
    } else if (keyCode == KeyEvent.KEYCODE_CAMERA) {
        toggleRecording();
        return true;
    } else {
        return super.onKeyDown(keyCode, event);
    }
}
```

Recording a Time-lapse

The methods in this section are responsible for recording the time-lapse.

1. Implement the `toggleRecording` method.

Tapping on the only menu item in the activity calls the `toggleRecording` method, which starts or stops recording a time-lapse. Note that this method plays a sound when a time-lapse starts or stops recording.

In `TimelapseActivity.java`:

```
private void toggleRecording() {
    if (mIsRecording) {
        // stop recording and release camera
        mMediaRecorder.stop(); // stop the recording
        releaseMediaRecorder(); // release the MediaRecorder object
        mCamera.lock();        // take camera access back from MediaRecorder

        CameraUtils.scanFile(this, mOutputFilename);

        // inform the user that recording has stopped
        playSound(mStopRecordingSoundId);
        mIsRecording = false;
    } else {
        // initialize video camera
        try {
            if (prepareVideoRecorder()) {
                // Camera is available and unlocked, MediaRecorder is prepared,
                // now you can start recording
                mMediaRecorder.start();

                // inform the user that recording has started
                playSound(mStartRecordingSoundId);
                mIsRecording = true;
            } else {
                // prepare didn't work, release the camera
                releaseMediaRecorder();
                // inform user
            }
        } catch (Exception e) {
            mCamera.release();
        }
    }
}
}
```

2. Initialize the `MediaRecorder` in the `prepareVideoRecorder` method.

This method is derived from <http://developer.android.com/guide/topics/media/camera.html>.

In `TimelapseActivity.java`:

```
private boolean prepareVideoRecorder(){
    mMediaRecorder = new MediaRecorder();

    // Step 1: Unlock and set camera to MediaRecorder
    mCamera.unlock();
    mMediaRecorder.setCamera(mCamera);

    // Step 2: Set sources
    //mMediaRecorder.setAudioSource(MediaRecorder.AudioSource.DEFAULT);
    mMediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
}
```



```

// Step 3: Set a CamcorderProfile
mMediaRecorder.setProfile(CamcorderProfile.get(CamcorderProfile.QUALITY_TIME_LAPSE_HIGH));
mMediaRecorder.setCaptureRate(0.5);

// Step 4: Set output file
mOutputFilename = CameraUtils.getOutputTimelapseFile().toString();
mMediaRecorder.setOutputFile(mOutputFilename);

// Step 5: Set the preview output
mMediaRecorder.setPreviewDisplay(mPreview.getHolder().getSurface());

// Step 6: Prepare configured MediaRecorder
try {
    mMediaRecorder.prepare();
} catch (IllegalStateException e) {
    Log.d(TAG, "IllegalStateException preparing MediaRecorder: " + e.getMessage());
    releaseMediaRecorder();
    return false;
} catch (IOException e) {
    Log.d(TAG, "IOException preparing MediaRecorder: " + e.getMessage());
    releaseMediaRecorder();
    return false;
}
return true;
}

```

The two most important lines of `prepareVideoRecorder` are

```

mMediaRecorder.setProfile(CamcorderProfile.get(CamcorderProfile.QUALITY_TIME_LAPSE_HIGH));
mMediaRecorder.setCaptureRate(0.5);

```

The first line configures `MediaRecorder` to record a time-lapse. The second line specifies the rate at which images are captured for the time-lapse. In particular, the capture rate is the inverse of the interval between frames in seconds. To record a time-lapse that captures a frame every two seconds, set the capture rate to $1/2$ or 0.5 .

3. Implement `releaseMediaRecorder`.

This method releases the `MediaRecorder` and frees the resources it uses.

In `TimelapseActivity.java`:

```

private void releaseMediaRecorder(){
    if (mMediaRecorder != null) {
        mMediaRecorder.reset(); // clear recorder configuration
        mMediaRecorder.release(); // release the recorder object
        mMediaRecorder = null;
        mCamera.lock(); // lock camera for later use
    }
}

```

At this point, run the example and verify that you can record time-lapses. Note that there are many ways in which this Glassware could be improved. For instance, you could

1. allow a user to select the interval between frames,
2. display feedback while recording a time-lapse (such as the duration of the recording), and
3. dim the screen after a certain period of inactivity while recording to conserve battery.

Additionally, you could try to record the time-lapse from a LiveCard.

Summary

In this chapter, we learned how to take pictures and record videos with the Glass camera. We captured media by leveraging the native camera app with intents and by using the camera API directly. Additionally, we utilized the camera API from a LiveCard and implemented an activity that can record time-lapses.

Part

V

Android Wear and Glass

Chapter 12

Location and Orientation

During everyday usage, people rarely focus their attention exclusively on wearables because their primary interests and tasks are part of the real world. As a result, wearables should minimize the amount of user intervention they require. In a way, user intervention is a limited resource, just like battery consumption or network bandwidth. The need for user intervention is reduced by knowing where users are located and in which direction their devices are facing. For instance, an application could remind users to carry their umbrellas as they leave their homes on a day with a rainy forecast. Location and orientation are the foundation for contextual awareness, and exploiting these capabilities is likely to lead to innovative wearable apps.

Apps may also respond to the state of the analog world by using location and orientation as an input. A compass app, for example, presents information relative to the world. Another example is a navigation app, which provides directions relative to a user's current position.

In this chapter, we'll learn how to mathematically represent location and orientation. Then we'll write code to obtain location and orientation updates for both Android Wear and Glass. Additionally, we'll write a compass app for both devices, and we'll learn to build a pedometer for Android Wear.

We'll start by discussing how the earth is modeled so we can properly understand the definition of latitude and longitude.

Modeling the Earth as an Ellipsoid

Even though the earth is "round," its shape isn't exactly spherical. The distance from the center of the earth to the North Pole, for instance, is shorter than the distance to the equator. The shape of the earth is typically modelled as an ellipsoid, which is also known as an oblate spheroid. An ellipsoid is the 3D shape that results from rotating an ellipse about its minor axis (see Figure 12-1).

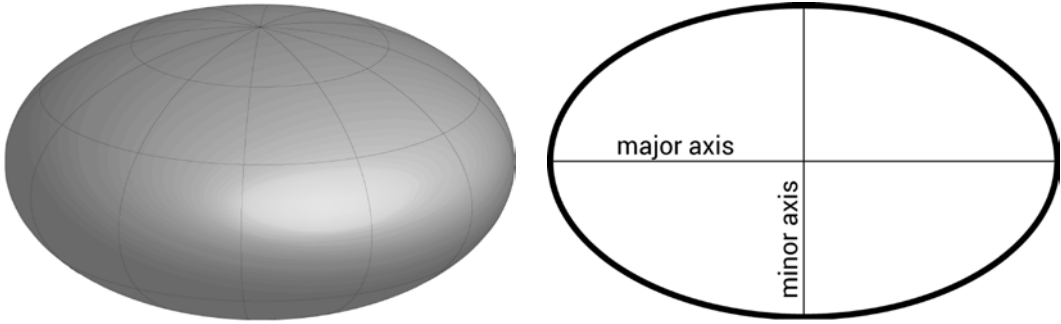


Figure 12-1. The shape of the earth is typically modeled as an ellipsoid (left). An ellipsoid is generated by revolving an ellipse about its minor axis, which is its smallest diameter (right)

Although the earth is not exactly ellipsoid because its surface is irregular and its mass is not distributed uniformly, modeling it as an ellipsoid results in a good overall fit that leads to accurate calculations.

The parameters that determine the shape and size of the specific ellipsoid that models the earth (which is known as a reference ellipsoid) are specified by a *datum*. Applications that use GPS utilize the WGS-84 datum, which defines the ellipsoids parameters as

$$a = 6,378,137.0 \text{ m}$$

$$b = 6,356,752.314245 \text{ m}$$

Where 'a' is half of the major axis and 'b' is half of the minor axis.

Geodetic Latitude and Longitude

The location of a point on the surface of the earth is uniquely determined by its latitude and longitude. There are multiple ways of defining latitude and longitude, and each way can yield different coordinates for a given point. Applications that use GPS coordinates utilize the conventions defined by geodetic latitude and longitude because they have certain mathematical properties that simplify many calculations. Before defining latitude and longitude, we must define a few other terms.

The *equatorial plane* is the plane that passes through the earth's equator as shown in Figure 12-2.

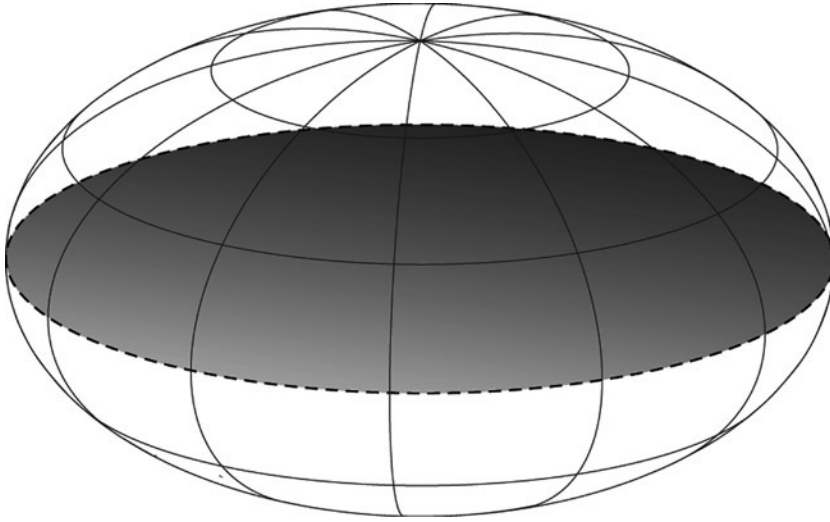


Figure 12-2. The equatorial plane intersects with the earth's equator and is perpendicular to the axis of rotation

Meridians of longitude are lines on the surface of the ellipsoid that connect the north and south poles and are perpendicular to the equatorial plane. The meridian of longitude that intersects with the Royal Observatory in Greenwich, London is known as the *prime meridian*. This meridian was arbitrarily selected as a reference that divides the globe into eastern and western hemispheres. Figure 12-3 shows several meridians of longitude.

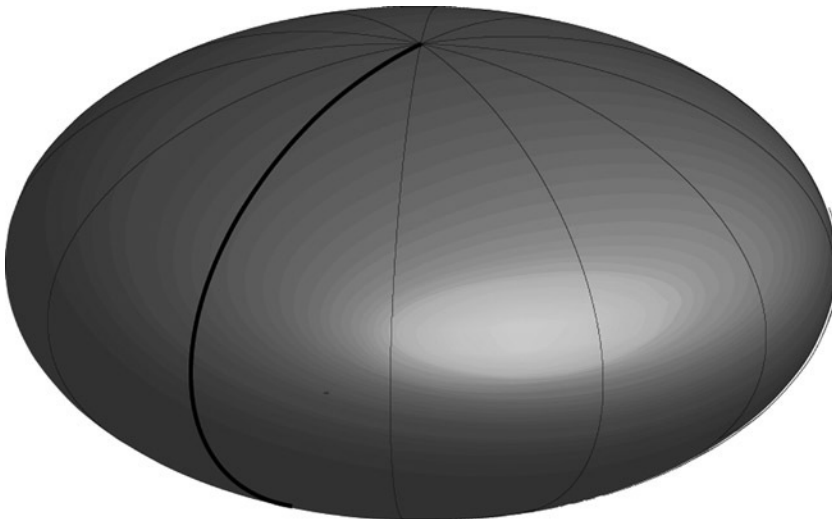


Figure 12-3. Several meridians of longitude are shown on an ellipsoid. The meridian with a thick line represents the prime meridian in this illustration

Now that we've defined the equatorial plane and meridians of longitude, we are ready to define geodetic latitude and longitude. The *geodetic longitude* of a point is the angle between the prime meridian and the meridian that intersects the point, as shown in Figure 12-4.

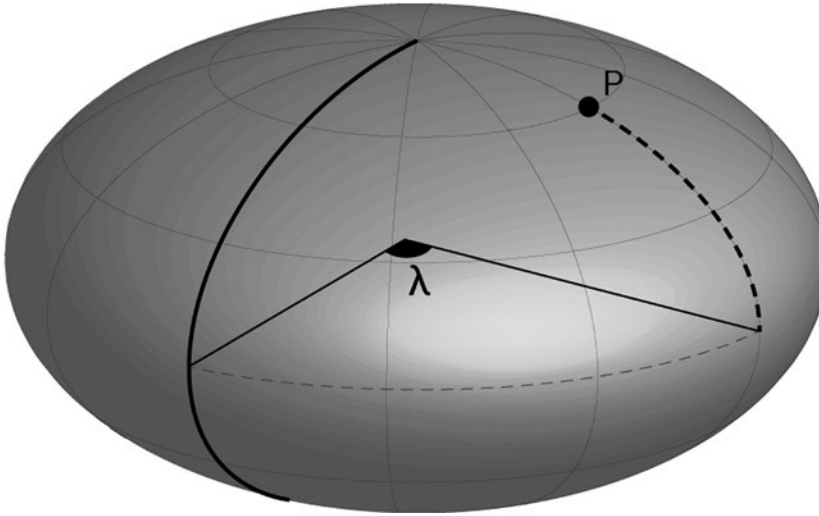


Figure 12-4. The longitude of point *P* is the angle between the prime meridian and the meridian that intersects *P*. Here, the prime meridian is highlighted by a dark line, the meridian that intersects *P* is shown as a dashed line, and the longitude of *P* is represented by lambda (λ)

The *geodetic latitude* of a point is the angle between the equatorial plane and the line that is normal (that is, perpendicular) to the surface of the earth at the point, as shown in Figure 12-5.

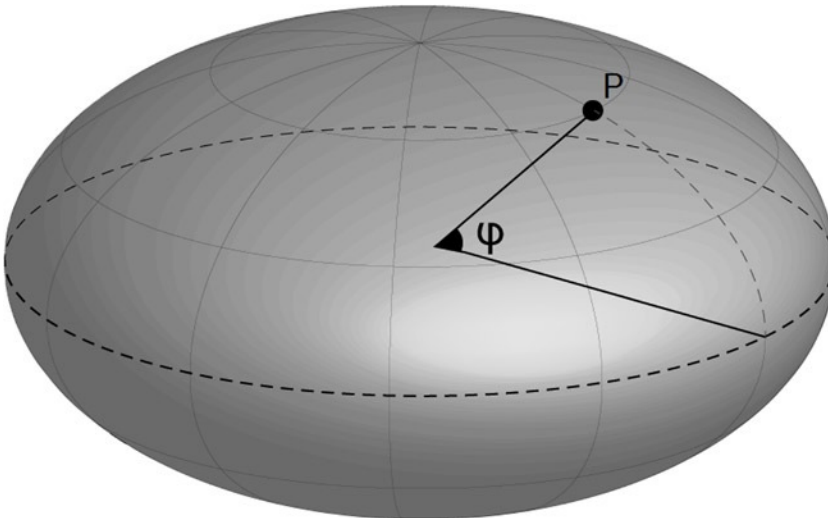


Figure 12-5. Geodetic latitude is the angle between the equatorial plane and the line that is perpendicular to the surface of the earth at a given point. Here, the equatorial plane is shown by a dashed line, and the latitude of point *P* is represented by the letter phi (ϕ)

Note that the line that is normal to a point on the surface of the earth does not necessarily intersect with the center of the earth, as illustrated by Figure 12-6. If the model of the earth were a perfect sphere, then the line normal to a point on the surface would always intersect with the center, but this property does not hold for ellipsoids.

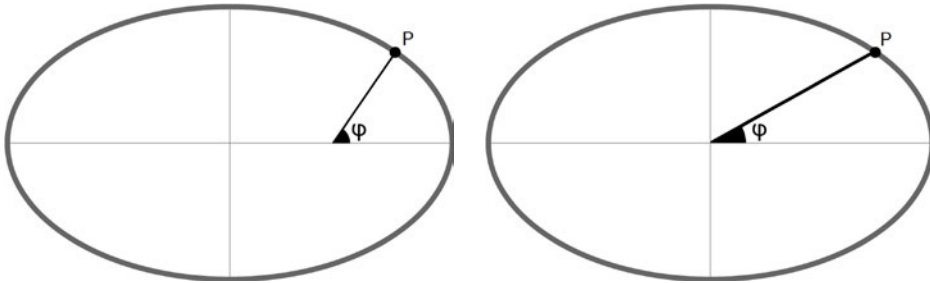


Figure 12-6. The line that defines geodetic latitude of a point *P* is perpendicular to the surface of the ellipsoid and does not necessarily intersect with the center of the earth, as depicted by the diagram on the left. The line that intersects with both the point *P* and the center of the earth may result in a different angle (right)

Throughout the rest of the book, I'll refer to geodetic latitude and longitude as simply latitude and longitude.

Defining Altitude

GPS represents altitude as the height above the reference ellipsoid in meters. The reference ellipsoid is only a model of the surface of the earth and does not correspond to the real surface of the earth. Do not assume that the altitude given by GPS is related to the height above the ground or to the height above sea level.

Location Providers

Android devices can obtain location updates from a variety of different sources, each of which has its own benefits and detriments. Each source is encapsulated into a location provider, and the most common providers are:

- **Network Provider:** determines the device's location based on the availability of cellular towers and WiFi networks. When available, this provider is fast and consumes low battery power. However, the accuracy of this provider is limited. Requires the `ACCESS_COARSE_LOCATION` permission.
- **GPS Provider:** utilizes GPS to obtain an accurate location estimate. This provider consumes a large amount of battery and may take some time to provide a location fix. Since GPS derives location based on signals from satellites, it does not work indoors or in areas without a direct line of sight to the sky. Requires the `ACCESS_FINE_LOCATION` permission.

- **Passive Provider:** does not initiate a location fix but passively receives any location updates that are initiated by other applications. As a result, this provider consumes a very low amount of battery but may not receive location updates for a long time. Requires the `ACCESS_FINE_LOCATION` permission even though it may receive coarse updates if the GPS provider is not enabled.

This list of location providers is not exhaustive or comprehensive. Glass, for instance, does not have a GPS provider and Android Wear devices do not have a network provider. These devices also have additional location providers that are responsible for fetching location updates from a mobile device.

Directly selecting a location provider is not a good practice because we have no way of knowing whether additional providers are available. The best way of selecting a provider is by delegating the decision to the fused location provider that is included with Google Play Services. This provider lets an app specify the accuracy and rate at which it needs location updates and obtains optimized location values from one or more providers. While Android Wear devices should always use the fused location provider, Glass does not yet support Google Play Services and must instead use the `Location` API directly.

In the next section, we'll learn to obtain location updates on Android Wear devices from the fused location provider. Then, we'll learn an alternative method to obtain location updates on Glass without using Google Play Services.

Obtaining Location Updates on Android Wear

Obtaining location on Android Wear devices has a few caveats:

- Android Wear devices can obtain location updates from the mobile devices they are paired to through the connection established with the companion app.
- Android Wear devices cannot connect to cellular or WiFi and, as a result, they cannot directly obtain location updates from the network provider. Future Android Wear devices may support WiFi, but for now we cannot rely on the network provider.
- Some watches such as the Sony SmartWatch 3 contain internal GPS units and can obtain location updates without a paired phone.
- Even if a watch has an internal GPS, apps should obtain location updates through the mobile device when possible to conserve the watch's battery.

Google Play Service's fused location provider will automatically fetch location from a paired mobile device if available. If no device is paired to a watch, fused location provider will attempt to obtain location updates from the watch's internal GPS unit, if available. If the watch does not have an internal GPS unit, fused location provider will be unable to obtain location updates.

Your app is responsible for displaying an error message when it detects that the watch is not paired to a mobile device and has no internal GPS.

In this example, we'll obtain location updates from the fused location provider and display them on the watch's screen. Additionally, we'll display an error message if the app is unable to obtain location updates.

Start by creating an activity called `WearLocationActivity`.

1. Create a layout.

This layout contains `TextViews` that display the current latitude, longitude, and altitude that are received from the fused location provider. Additionally, the layout contains a `TextView` that indicates whether the watch is paired to a handheld device.

In res ► layout ► `activity_location.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:padding="8dip"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        app:layout_box="top|left"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:text="Latitude"
            android:textColor="#808080"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <TextView
            android:id="@+id/latitude"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <TextView
            android:text="Longitude"
            android:textColor="#808080"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <TextView
            android:id="@+id/longitude"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
```

```

<TextView
    android:text="Altitude"
    android:textColor="#808080"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/altitude"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/warning"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

</LinearLayout>

</android.support.wearable.view.BoxInsetLayout>

```

2. Add the ACCESS_FINE_LOCATION permission.

In AndroidManifest.xml:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

3. Declare WearLocationActivity as a launcher activity so it can be started with an app-provided voice action.

In WearLocationActivity.java:

```

<activity android:name=".WearLocationActivity"
    android:taskAffinity=
        "com.ocddevelopers.androidwearables.locationandorientation.WearLocation"
    android:label="Wearable Location">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

```

4. Declare constants and member variables.

In WearLocationActivity.java:

```

public class WearLocationActivity extends Activity {
    private static final int UPDATE_INTERVAL_MS = 500;
    private static final int FASTEST_INTERVAL_MS = 500;
    private static final String TAG = "Wear";
    private GoogleApiClient mGoogleApiClient;
    private TextView mLatitudeText, mLongitudeText, mAltitudeText, mWarningText;
    ...
}

```

5. In `onCreate` instantiate `GoogleApiClient` and add the `LocationServices` and `Wearable` APIs.

To access the fused location provider, which is part of the `LocationServices` API, the app must connect to Google Play Services through `GoogleApiClient`. This process is almost identical to using the wearable data layer API, as we saw in Chapter 6. Speaking of the data layer API, this example uses the Node API to check if the watch is paired to a mobile device.

In `WearableLocationActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_location);

    mLatitudeText = (TextView) findViewById(R.id.latitude);
    mLongitudeText = (TextView) findViewById(R.id.longitude);
    mAltitudeText = (TextView) findViewById(R.id.altitude);
    mWarningText = (TextView) findViewById(R.id.warning);

    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addApi(LocationServices.API)
        .addApi(Wearable.API)
        .addConnectionCallbacks(mConnectionCallbacks)
        .addOnConnectionFailedListener(mConnectionFailedListener)
        .build();
}
```

Register for Node and Location Updates

1. The `onConnected` method will be called once the app successfully connects to Google Play Services. Here, request location updates from the fused location provider and register a `NodeListener` to get notified when a handheld device is paired or unpaired.

`NodeListener` only notifies the app if a handheld connects to or disconnects from the current device. It does not notify the app whether the handheld device is connected or disconnected to begin with. Use the Node API's `getConnectedNodes` method to see if there are any devices connected to the watch. Assume that the mobile device is unpaired if there are no devices connected to the watch.

In `WearableLocationActivity.java`:

```
private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks =
    new GoogleApiClient.ConnectionCallbacks() {
    @Override
    public void onConnected(Bundle bundle) {
        requestLocationUpdates();

        Wearable.NodeApi.addListener(mGoogleApiClient, mNodeListener);
    }
}
```

```

Wearable.NodeApi.getConnectedNodes(mGoogleApiClient).setResultCallback(
    new ResultCallback<NodeApi.GetConnectedNodesResult>() {
        @Override
        public void onResult(NodeApi.GetConnectedNodesResult result) {
            if(result.getNodes().size() == 0) {
                // the watch is not paired to the Android Wear device
                updateWarning(false);
            }
        }
    });
}

@Override
public void onConnectionSuspended(int i) {
}
};

```

The `updateWarning` method, which is implemented shortly, displays a message when the watch is not paired to a handheld device.

Connect to and disconnect from `GoogleApiClient` in `onResume` and `onPause`, respectively.

In `WearableLocationActivity.java`:

```

@Override
protected void onResume() {
    super.onResume();
    mGoogleApiClient.connect();
}

@Override
protected void onPause() {
    mGoogleApiClient.disconnect();
    removeLocationUpdates();
    super.onPause();
}

```

The `requestLocationUpdates` and `removeLocationUpdates` methods are implemented below.

2. Implement `NodeListener`.

The `NodeListener` instance gets notified when a mobile device connects or disconnects to or from the watch. If a mobile device is not connected and the watch does not have an internal GPS, we display a message to notify the user. If a mobile device connects to the watch, we clear the error message.

In `WearableLocationActivity.java`:

```

private NodeApi.NodeListener mNodeListener = new NodeApi.NodeListener() {
    @Override
    public void onPeerConnected(Node node) {
        updateWarning(true);
    }
}

```

```

@Override
public void onPeerDisconnected(Node node) {
    updateWarning(false);
}
};

```

3. Implement the `hasGps` method, which returns true if the watch has an internal GPS.

In `WearableLocationActivity.java`:

```

private boolean hasGps() {
    return getPackageManager().hasSystemFeature(PackageManager.FEATURE_LOCATION_GPS);
}

```

4. Implement the `updateWarning` method, which displays a message that warns the user when the watch cannot obtain location updates because 1) it's not paired with a handheld device and 2) it does not have an internal GPS.

In `WearableLocationActivity.java`:

```

private void updateWarning(final boolean phoneAvailable) {
    // ensure that mWarningText.setText is called from the UI thread
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            String message = "";
            if(!phoneAvailable && !hasGps()) {
                message = "GPS unavailable. Pair watch to the phone to resume.";
            }

            mWarningText.setText(message);
        }
    });
}

```

Before completing this example's implementation, let's discuss `FusedLocationProvider`.

Using `FusedLocationProvider`

When requesting location updates, the fused location provider requires apps to specify how quickly they need location updates and whether they prefer higher accuracy or power consumption. These parameters are specified in a `LocationRequest` object.

LocationRequest Priority

The priority of `LocationRequest` specifies whether higher accuracy or power consumption is desirable. It can take the following values:

- `PRIORITY_NO_POWER`: essentially uses the passive location provider to get location updates only when they are requested by other apps.
- `PRIORITY_LOW_POWER`: reduces battery consumption by requesting city level accuracy (that is, about 10km).
- `PRIORITY_BALANCED_POWER_ACCURACY`: requests updates with an accuracy of about 100m.
- `PRIORITY_HIGH_ACCURACY`: requests the most accurate location updates and does not attempt to minimize power consumption.

Interval and FastestInterval

These parameters specify how quickly the app should receive location updates. The `setInterval` method specifies the desired interval for location updates. This interval is merely a suggestion and the system may provide updates at a slower rate, if location updates are unavailable, or at a faster rate, if another app has requested faster updates. An interval of 0 indicates that the updates should occur as fast as possible, but doing so is not recommended since future APIs may trigger location updates at an extremely fast rate. A fast and reasonable interval is 500 ms.

The `fastestInterval` specifies the maximum interval at which location updates can be triggered. Unlike the `interval`, this parameter is not a suggestion—updates cannot occur at a faster rate under any circumstance. The official documentation also recommends that this value is not set to 0 for the same reason `Interval` should not be set to 0.

Requesting and Removing Location Updates

Now that we understand what parameters form a `LocationRequest`, let's implement the code that requests and removes location updates.

1. Implement the `requestLocationUpdates` method, which requests location updates from the fused location provider.

In this example, we request high-accuracy updates with an `Interval` and a `FastestInterval` of 500 milliseconds with the Fused Location API, which can be accessed from the `LocationServices.FusedLocationApi` class. This class's `requestLocationUpdates` method can be executed synchronously or asynchronously just like other APIs in Google Play Services. Here, we execute the method asynchronously by calling the `setResultCallback` method. In this callback we simply log whether the callback was set successfully, which should be the case unless you are missing a permission.

In `WearableLocationActivity.java`:

```
private void requestLocationUpdates() {
    LocationRequest locationRequest = LocationRequest.create()
        .setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY)
        .setInterval(UPDATE_INTERVAL_MS)
        .setFastestInterval(FATEST_INTERVAL_MS);

    LocationServices.FusedLocationApi
        .requestLocationUpdates(mGoogleApiClient, locationRequest, mLocationListener)
        .setResultCallback(new ResultCallback() {

            @Override
            public void onResult(Result result) {
                Status status = result.getStatus();
                if (status.getStatus().isSuccess()) {
                    Log.d(TAG, "Successfully requested location updates");
                } else {
                    Log.e(TAG,
                        "Failed in requesting location updates, "
                        + "status code: "
                        + status.getStatusCode()
                        + ", message: "
                        + status.getStatusMessage());
                }
            }
        });
}
```

2. Implement the `removeLocationUpdates` method.

This method asks the fused location provider to stop sending GPS updates.

In `WearableLocationActivity.java`:

```
private void removeLocationUpdates() {
    if (mGoogleApiClient.isConnected()) {
        LocationServices.FusedLocationApi
            .removeLocationUpdates(mGoogleApiClient, mLocationListener);
    }
}
```

3. Implement `LocationListener`, which receives location updates from the fused location provider and displays them on the screen.

In `WearableLocationActivity.java`:

```
private LocationListener mLocationListener = new LocationListener() {
    @Override
    public void onLocationChanged(Location location) {
        mLatitudeText.setText(String.format(Locale.US, "%.6f", location.getLatitude()));
        mLongitudeText.setText(String.format(Locale.US, "%.6f", location.getLongitude()));
        mAltitudeText.setText(String.format(Locale.US, "%.2f", location.getAltitude()));
    }
};
```


At this point, run the app and you should see an output similar to Figure 12-7 (with different values of latitude and longitude).

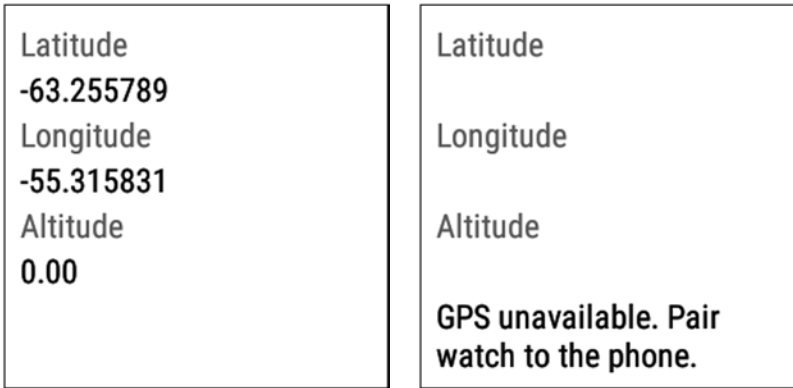


Figure 12-7. The output of `WearLocationActivity`. The screen displays the current location when location is available (left), and an error message notifies the user when location is not available (right)

Obtaining Location Updates on Glass

As of version XE 22, Glass does not support Google Play Services. As a result, Glassware cannot utilize the Fused Location Provider API and must instead obtain location updates using `LocationManager`. This example demonstrates how to use `LocationManager` to obtain location updates on Glass by displaying the current latitude, longitude, and altitude.

Begin by creating an activity called `GpsActivity`.

1. Create a voice trigger with the “Demo GPS” keyword.

In `res > xml > gps_trigger.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Demo GPS" />
```

2. Since we're using an unlisted voice command, add the development permission. Also add the access fine location permission since we want to receive location updates.

In `AndroidManifest.xml`:

```
<uses-permission android:name="com.google.android.glass.permission.DEVELOPMENT" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

3. Configure the activity to be started in response to the voice command defined above.

In `AndroidManifest.xml`:

```
<activity
    android:name=".GpsActivity">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>

    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/gps_trigger" />
</activity>
```

4. Create a layout.

This layout contains `TextViews` that display the current latitude, longitude, and altitude.

In `res > values > styles.xml`:

```
<style name="GlassTextSmall">
    <item name="android:fontFamily">sans-serif-light</item>
    <item name="android:textSize">32px</item>
    <item name="android:lineSpacingExtra">8px</item>
</style>

<style name="GlassTextNormal">
    <item name="android:fontFamily">sans-serif-light</item>
    <item name="android:textSize">40px</item>
    <item name="android:lineSpacingExtra">8px</item>
</style>
```

In `res > layout > activity_gps.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:columnCount="2"
    android:paddingTop="40px"
    android:paddingLeft="40px"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:text="Latitude"
        style="@style/GlassTextSmall"
        android:textColor="#808080"
        android:layout_marginRight="20px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

```
<TextView
    android:id="@+id/latitude"
    style="@style/GlassTextNormal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:text="Longitude"
    style="@style/GlassTextSmall"
    android:textColor="#808080"
    android:layout_marginRight="20px"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/longitude"
    style="@style/GlassTextNormal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:text="Altitude"
    style="@style/GlassTextSmall"
    android:textColor="#808080"
    android:layout_marginRight="20px"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/altitude"
    style="@style/GlassTextNormal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

```
</GridLayout>
```

5. Declare member variables.

In `GpsActivity.java`:

```
public class GpsActivity extends Activity {
    private LocationManager locationManager;
    private TextView mLatitudeText, mLongitudeText, mAltitudeText;
    ...
}
```

6. Initialize member variables.

Wrap the layout with `TuggableView` to give users tugging feedback. `TuggableView` is available in the sample source code and was covered in Chapter 8.

In GpsActivity.java:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    super.onCreate(savedInstanceState);
    View content = getLayoutInflater().inflate(R.layout.activity_gps, null);
    setContentView(new TuggableView(this, content));

    mLocationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    mLatitudeText = (TextView) content.findViewById(R.id.latitude);
    mLongitudeText = (TextView) content.findViewById(R.id.longitude);
    mAltitudeText = (TextView) content.findViewById(R.id.altitude);
}
```

7. Notify the user that tapping on the touchpad has no effect by playing the Sounds.DISALLOWED sound.

Recall that Glass automatically translates certain gestures into D-pad key events. Namely, tapping the touchpad triggers a key event with a key code of KEYCODE_DPAD_CENTER.

In GpsActivity.java:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if(keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
        ((AudioManager) getSystemService(Context.AUDIO_SERVICE))
            .playSoundEffect(Sounds.DISALLOWED);
        return true;
    } else {
        return super.onKeyDown(keyCode, event);
    }
}
```

Request and Remove Location Updates

In a previous section, we enumerated three location providers that are typically found in Android devices: the network provide, the GPS provider, and the passive provider. Glass, in contrast, lacks a GPS provider but contains a remote network provider and a remote GPS provider. These remote providers obtain location from a paired handheld device's network and GPS providers, respectively.

To obtain the most reliable location updates, select the best location providers with the `getProviders` method of the `LocationManager` class, which takes into account the accuracy of location updates that your app needs to obtain a list of the best available providers. Specify the accuracy of location updates that your app needs with the `Criteria` class, which contains a property called "accuracy" that can take the following values:

- **Criteria.ACCURACY_COARSE:** approximate location updates are sufficient. These may be obtained through a network provider.
- **Criteria.ACCURACY_FINE:** finer location updates are needed. These are usually obtained through GPS.

1. In the `requestGpsUpdates` method, obtain a list of the best location providers and request updates from all of them.

In this example, we'll request fine location updates by specifying an accuracy of `Criteria.ACCURACY_FINE`. Once we get a list of providers, we'll request location updates from all of them.

In `GpsActivity.java`:

```
private void requestGpsUpdates() {
    Criteria criteria = new Criteria();
    criteria.setAccuracy(Criteria.ACCURACY_FINE);
    criteria.setAltitudeRequired(true);

    List<String> providers = mLocationManager.getProviders(criteria, true);

    for (String provider : providers) {
        mLocationManager.requestLocationUpdates(provider, 0,
            0, mLocationListener);
    }
}
```

2. Implement the `removeGpsUpdates` method.

In `GpsActivity.java`:

```
private void removeGpsUpdates() {
    mLocationManager.removeUpdates(mLocationListener);
}
```

3. When a location update is received, display the coordinates on the screen.

In `GpsActivity.java`:

```
private LocationListener mLocationListener = new LocationListener() {
    @Override
    public void onLocationChanged(Location location) {
        mLatitudeText.setText(String.format(Locale.US, "%.6f", location.getLatitude()));
        mLongitudeText.setText(String.format(Locale.US, "%.6f", location.getLongitude()));
        mAltitudeText.setText(String.format(Locale.US, "%.2f", location.getAltitude()));
    }

    @Override
    public void onStatusChanged(String provider, int status, Bundle extras) {
    }

    @Override
    public void onProviderEnabled(String provider) {
    }
}
```

```
@Override
public void onProviderDisabled(String provider) {
}
};
```

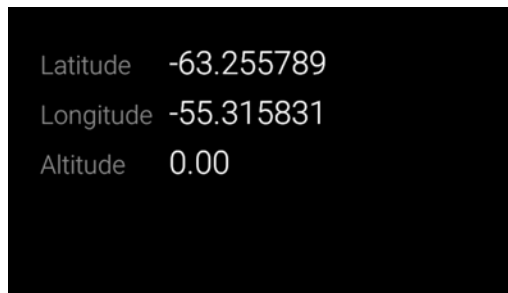
4. Ensure that location updates are only received when the app is in the foreground by requesting them in `onResume` and removing them in `onPause`.

In `GpsActivity.java`:

```
@Override
protected void onResume() {
    super.onResume();
    requestGpsUpdates();
}
```

```
@Override
protected void onPause() {
    removeGpsUpdates();
    super.onPause();
}
```

Now, run the app and verify that you obtain location updates as shown in Figure 12-8.



```
Latitude -63.255789
Longitude -55.315831
Altitude 0.00
```

Figure 12-8. The output of `GpsActivity` on Glass

Representing Orientation

Most Android devices, including Glass and Wear, contain sensors that can measure their orientation with respect to the world. These sensors, for instance, allow mobile devices to switch between portrait and landscape mode when users rotate their devices, and they help Android Wear devices figure out when to enable interactive mode. Additionally, applications such as a compass and a constellation tracker rely on these sensors to display information relative to the world.

Accessing the orientation of a device is relatively straightforward with regards to programming, but the concepts and mathematics of orientation are surprisingly involved. While the Android SDK encapsulates many complicated mathematical operations, we must understand several concepts before we can utilize them correctly.

This section will discuss

1. the three types of sensors used to measure a device's orientation,
2. the mathematical representation of orientation, and
3. how to obtain the orientation of a device in code.

There are several different mathematical representations of orientation, each of which has its own benefits and detriments. The most common representation, which is called *azimuth, pitch, and roll*, utilizes three angles to describe the orientation of an object. Each angle describes an object's rotation about a particular axis. To understand this representation, we must first define two different coordinate frames.

Coordinate Frames

In this context, a coordinate frame, also known as a coordinate system, defines the position of three axes that serve as a reference for an object's orientation. In particular, there are two coordinate frames of relevance.

The Inertial or Device Coordinate Frame

The device coordinate frame is attached to the center of the device and defines the x, y, and z axes as shown in Figure 12-9.

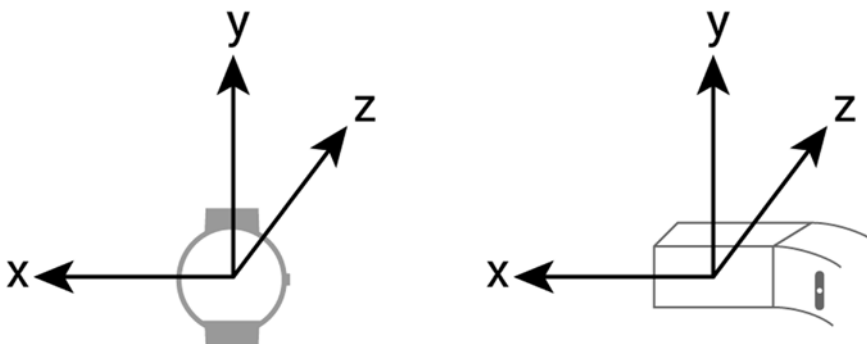


Figure 12-9. The device coordinate frame is attached to the center of a device. This figure shows the device frame with respect to an Android Wear device (left) and Glass (right)

The device coordinate frame moves and rotates along with the device.

The Earth or World Coordinate Frame

The world coordinate frame is defined with respect to a device's location on earth and does not move as the device rotates. In particular,

- the Y-axis is tangential to the surface of the earth and directed towards magnetic north.
- the Z-axis is perpendicular to the surface of the earth and points towards the center of the earth.
- the X-axis is perpendicular to both the Y and the Z axes and is approximately directed towards magnetic west.

Note In these definitions, the surface of the earth does not literally refer to the surface of the ground, but rather to the surface of the reference ellipsoid at a given location. If a device is located on a steep mountain, for example, the Z-axis of the world coordinate frame is not perpendicular to the mountain, but to the reference ellipsoid that models the surface of the earth as a whole.

Note that the world coordinate frame depends on the current location of the device since directions are relative to earth. For instance, the direction of “east” near the equator is almost perpendicular to the direction of “east” near the North Pole. However, we can assume that a world coordinate frame does not change unless users travel extremely large distances.

This frame is illustrated in Figure 12-10.

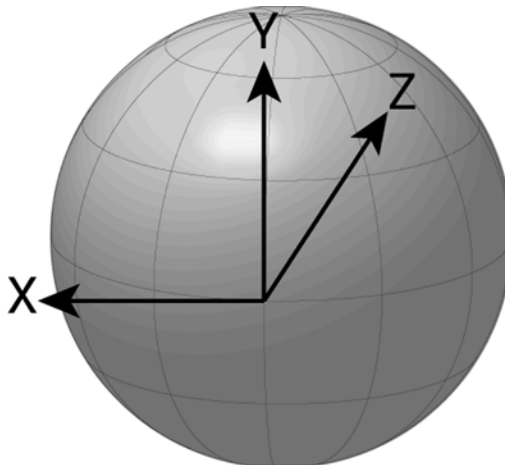


Figure 12-10. The world frame is defined for a device's current location with respect to the earth

Azimuth, Pitch, and Roll

These three angles describe the orientation of a device by transforming the world coordinate frame to the device coordinate frame for a given orientation.

Begin by aligning the with the world coordinate frame (X, Y, Z) as shown in Figure 12-11.

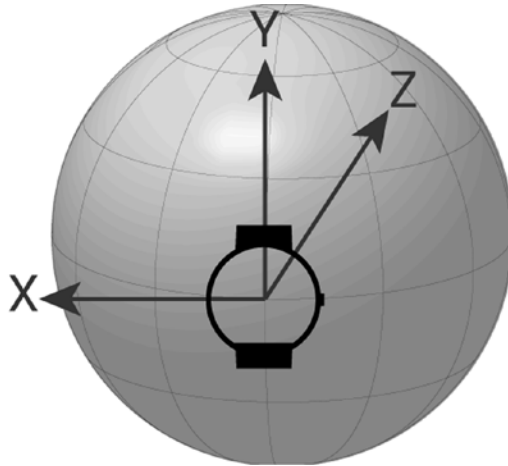


Figure 12-11. Aligning an Android Wear device to the world coordinate frame

Then, rotate the device about the Z-axis by the azimuth (labelled ϕ) to obtain a new coordinate frame (X_1, Y_1, Z_1) as illustrated by Figure 12-12.

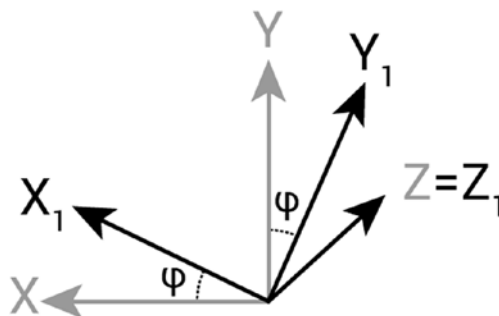


Figure 12-12. Rotating about the Z-axis by the azimuth angle

Note The direction of positive rotation is defined by the right-hand rule. If you are unfamiliar with this rule, I recommend searching for an explanation on YouTube since this concept is best illustrated with a video.

Next, rotate the device about the X_1 -axis by the pitch (labelled θ) to obtain a new coordinate frame (X_2, Y_2, Z_2) as illustrated by Figure 12-13.

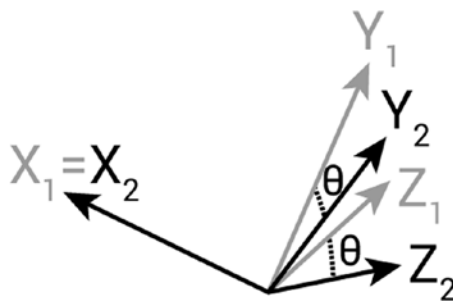


Figure 12-13. Rotating about the X_1 -axis by the pitch angle

Rotate the device about the Y_2 -axis by the roll (labelled Ψ) to obtain the device coordinate frame (x, y, z) as illustrated by Figure 12-14.

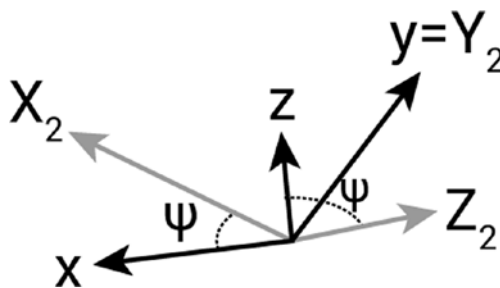


Figure 12-14. Rotating about the Y_2 -axis by the roll angle

Note that each rotation is applied about the coordinate frame defined by the previous rotation. The resulting coordinate frame is the device frame of the device at the orientation given by the azimuth, pitch, and roll.

There are many caveats regarding azimuth, pitch, and roll.

First, the order of the three rotations is not commutative. That is, rotating about azimuth, then pitch, and then roll is not equivalent to rotating about pitch, then azimuth, and then roll or any other order for that matter. The coordinate frames and the order described above is used by the Android SDK, but keep in mind that other disciplines may have different conventions.

Second, other disciplines may use different names for azimuth, pitch, and roll. For instance, azimuth may also be referred to as heading or yaw. There are no standards or conventions so these terms are not consistently defined.

Third, with the conventions described above, azimuth, pitch, and roll becomes finicky and unpredictable when the pitch has values close to 90 or -90 degrees. This problem is not related to the hardware sensors but to the mathematics behind azimuth, pitch, and roll. A detailed explanation of gimbal lock is outside the scope of this book, but its ramifications are straightforward: don't trust the orientation when pitch is close to +/- 90 degrees. If these values of pitch are unavoidable for an application, consider using rotation matrices or quaternions.

Fourth, there are multiple different ways of defining a device frame and a coordinate frame. The coordinate frame used by rotation matrices in Android, for instance, is different. The frames we defined in this chapter are utilized by the `SensorManager.getOrientation` method, which we'll use in the example code. The values of azimuth, pitch, and roll obtained with this method correspond to the rotations defined by these frames.

Rotation Matrices and Quaternions

The advantage of azimuth, pitch, and roll is that they can be intuitively understood. You can obtain a general idea of an object's orientation by taking a phone or any other object and manually subjecting it to the three rotations.

The main disadvantage of azimuth, pitch, and roll is that it suffers from gimbal lock. For applications in which facing directly up or down are unavoidable, azimuth, pitch, and roll cannot be used. For instance, an app for finding constellations such as Google Sky Maps must allow users to look directly above. These cases must rely on other coordinate systems that do not suffer from gimbal lock. Two popular alternatives are rotation matrices and quaternions. The mathematics behind these two coordinate systems are outside the scope of this book, but if you ever run into an application that cannot tolerate gimbal lock, consider researching rotation matrices and quaternions.

Measuring Orientation

There are three different types hardware sensors involved in measuring orientation:

- accelerometers measure the acceleration force that the sensor is subjected to. Acceleration forces can be static, such as the force of gravity, or dynamic, such as linear movement.
- gyroscopes measure angular velocity relative to the sensor and are not sensitive to linear acceleration.
- magnetometers measure magnetic fields and can infer the azimuth of the device based on the magnetic field of the earth.

By itself, each sensor cannot adequately measure a device's orientation. However, the measurements of all three sensors can be combined to calculate a device's orientation effectively. Accelerometers can measure the pitch and roll of the device based on the direction of gravity, but they cannot measure azimuth. Additionally, when accelerometers measure pitch and roll, they tend to amplify the jitter that naturally occurs when people hold a device. A filter can reduce the jitter at the expense of added latency. In other words, accelerometers can measure a device's pitch and roll with a noticeable latency.

Gyroscopes cannot directly measure a device's orientation, but they can calculate changes in orientation based on angular velocity. These changes in orientation can be used to calculate orientation relative to the initial position. Gyroscopes can be used to calculate relative orientation with a very low latency, but they suffer from drift. Drift is the accumulation of small errors in measuring changes in orientation and can lead to large errors in orientation.

Magnetometers can measure azimuth with a noticeable latency. That is, if the device quickly changes azimuth, the magnetometer will take several hundred milliseconds to measure the new value. These sensors are susceptible to magnetic interference.

By combining the output of these three sensors, a device can measure its orientation with little latency and without drift. The algorithm that merges the output of all sensors is called a Kalman filter and it works by trusting the gyroscope's measurements in the short run (since it has a low latency), the accelerometer's measurements of pitch and roll in the long run (since they don't suffer from drift), and the magnetometers measurement of azimuth in the long run (since it also doesn't suffer from drift).

Note The Moto 360 smartwatch does not have a magnetometer. As a result, its azimuth measurements are not measured relative to magnetic north, but relative to some arbitrary direction. Furthermore, the azimuth measurement drifts over time.

The Rotation Vector Sensor

Android supports a variety of hardware and software sensors. Hardware sensors obtain their measurements directly from sensors that reside within the Android device. In contrast, software sensors derive their measurements from a variety of sources, including hardware sensors.

In my opinion, the most useful sensor is the rotation vector sensor. This software sensor combines the outputs of a device's accelerometer, gyroscope, and magnetometer to obtain its current orientation. Although the rotation vector sensor outputs orientation in a strange format, it can easily be converted to azimuth, pitch, and roll.

Since the mathematics to convert a rotation vector to azimuth, pitch, and roll are tedious, the Android SDK encapsulates them in a few utility methods. We'll demonstrate how to use these methods both in Android Wear and in Glass.

Rotation Vector in Android Wear

After registering for rotation vector updates (as we'll see in the next section), sensor updates are passed to the `onSensorChanged` method of the `SensorEventListener` callback. The `SensorManager` class contains a method that converts the rotation vector into a rotation matrix, and another method that converts a rotation matrix to azimuth, pitch, and roll. These methods are `SensorManager.getRotationMatrixFromVector` and `SensorManager.getOrientation`, respectively. By chaining these two methods, we can convert a rotation vector to azimuth, pitch, and roll.

```
private float[] mRotationMatrix = new float[16];
private float[] mOrientation = new float[3];

@Override
public void onSensorChanged(SensorEvent event) {
    if(event.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR) {
        // convert rotation vector to azimuth, pitch, & roll
        SensorManager.getRotationMatrixFromVector(mRotationMatrix, event.values);
        SensorManager.getOrientation(mRotationMatrix, mOrientation);
        float azimuthDeg = (float) Math.toDegrees(mOrientation[0]);
        float pitchDeg = (float) Math.toDegrees(mOrientation[1]);
        float rollDeg = (float) Math.toDegrees(mOrientation[2]);
        ...
    }
}
```

The angles output by `SensorManager.getOrientation` are given in radians, and we should convert them to degrees if needed.

Rotation Vector in Glass

Converting a rotation vector to azimuth, pitch, and roll on Glass is similar to Android Wear, with one exception: we must compensate for Glass's slightly different coordinate system. We want azimuth to change when users move their heads from left to right. Similarly, we want pitch to change as users look up and down. To ensure that the values of azimuth, pitch, and roll obtained by Glass match these expectations, we need to remap the coordinate system. Fortunately, the Android SDK encapsulates this process, which is normally mathematically tedious.

```
@Override
public void onSensorChanged(SensorEvent event) {
    if(event.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR) {
        // convert rotation vector to azimuth, pitch, & roll
        SensorManager.getRotationMatrixFromVector(mRotationMatrix , event.values);

        // take into account Glass's coordinate system
        SensorManager.remapCoordinateSystem(mRotationMatrix, SensorManager.AXIS_X,
            SensorManager.AXIS_Z, mRotationMatrix);

        SensorManager.getOrientation(mRotationMatrix, mOrientation);
        float azimuthDeg = (float) Math.toDegrees(mOrientation[0]);
        float pitchDeg = (float) Math.toDegrees(mOrientation[1]);
        float rollDeg = (float) Math.toDegrees(mOrientation[2]);
    }
}
```

The `SensorManager.remapCoordinateSystem` method does all the heavy lifting. Otherwise, the rest of this procedure is identical to the one for Android Wear.

Implementing Rotation Vector in Android Wear

This example obtains updates from the rotation vector sensor, converts them to azimuth, pitch, and roll, and displays them on the screen.

1. Create a layout.

This layout contains `TextViews` that display the values of azimuth, pitch, and roll. The top level layout is a `BoxInsetLayout` to ensure that all fields are visible on both square and round devices.

In res ► layout ► activity_orientation.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:padding="8dip"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        app:layout_box="top|left"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:text="Azimuth"
            android:textColor="#808080"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <TextView
            android:id="@+id/azimuth"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <TextView
            android:text="Pitch"
            android:textColor="#808080"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <TextView
            android:id="@+id/pitch"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <TextView
            android:text="Roll"
            android:textColor="#808080"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <TextView
            android:id="@+id/roll"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
    </LinearLayout>
</android.support.wearable.view.BoxInsetLayout>
```

```

        <TextView
            android:id="@+id/warning"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

    </LinearLayout>

</android.support.wearable.view.BoxInsetLayout>

```

2. Declare `OrientationActivity` as a launcher activity so it can be started with an app-provided voice action.

In `AndroidManifest.xml`:

```

<activity android:name=".OrientationActivity"
    android:taskAffinity=
        "com.ocddevelopers.androidwearables.locationandorientation.WearOrientation"
    android:label="Orientation Demo">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

```

3. Start by declaring and initializing the member variables that we'll use throughout this example.

In `OrientationActivity.java`:

```

public class OrientationActivity extends Activity {
    private static final float TOO_STEEP_PITCH_DEGREES = 70.0f;
    private boolean mLowAccuracy, mTooSteep;
    private TextView mAzimuthText, mPitchText, mRollText, mWarningText;
    private SensorManager mSensorManager;
    private Sensor mRotationVectorSensor, mMagneticSensor;
    private float[] mRotationMatrix = new float[16];
    private float[] mOrientation = new float[3];
    ...
}

```

4. Initialize member variables.

The last three lines in the `onCreate` method obtain both the default rotation vector sensor and the default magnetic field sensor. The rotation vector sensor is used to measure orientation while the magnetic field sensor is only used to obtain an estimate of whether the measurement is accurate. We must check accuracy with the magnetic field sensor because the rotation vector sensor does not provide this information.

In `OrientationActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_orientation);

    mAzimuthText = (TextView) findViewById(R.id.azimuth);
    mPitchText = (TextView) findViewById(R.id.pitch);
    mRollText = (TextView) findViewById(R.id.roll);
    mWarningText = (TextView) findViewById(R.id.warning);

    mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    mRotationVectorSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR);
    mMagneticSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
}
```

5. Register and unregister for sensor updates in `onResume` and `onPause`, respectively, to ensure that the app only receives updates when the activity is in the foreground.

In `OrientationActivity.java`:

```
@Override
protected void onResume() {
    super.onResume();
    registerForSensorUpdates();
}

private void registerForSensorUpdates() {
    mSensorManager.registerListener(mSensorEventListener, mRotationVectorSensor,
        SensorManager.SENSOR_DELAY_UI);

    // obtain accuracy updates from the magnetic field sensor since the rotation vector
    // sensor does not provide any
    if(mMagneticSensor != null) {
        mSensorManager.registerListener(mSensorEventListener, mMagneticSensor,
            SensorManager.SENSOR_DELAY_UI);
    }
}

@Override
protected void onPause() {
    mSensorManager.unregisterListener(mSensorEventListener);
    super.onPause();
}
```

6. Handle sensor updates.

Sensor updates are passed to the `SensorEventListener` instance. In the `onSensorChanged` method, display the azimuth, pitch, and roll values in `TextViews`. Additionally, notify the user when the accuracy of the magnetic sensor decreases or when the pitch is too close to +/- 90 degrees.

In `OrientationActivity.java`:

```
private SensorEventListener mSensorEventListener = new SensorEventListener() {
    @Override
    public void onSensorChanged(SensorEvent event) {
        if(event.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR) {
            // convert rotation vector to azimuth, pitch, & roll
            SensorManager.getRotationMatrixFromVector(mRotationMatrix , event.values);

            SensorManager.getOrientation(mRotationMatrix, mOrientation);
            float azimuthDeg = (float) Math.toDegrees(mOrientation[0]);
            float pitchDeg = (float) Math.toDegrees(mOrientation[1]);
            float rollDeg = (float) Math.toDegrees(mOrientation[2]);

            mAzimuthText.setText(String.format(Locale.US, "%.1f", azimuthDeg));
            mPitchText.setText(String.format(Locale.US, "%.1f", pitchDeg));
            mRollText.setText(String.format(Locale.US, "%.1f", rollDeg));

            mTooSteep = pitchDeg > TOO_STEEP_PITCH_DEGREES ||
                pitchDeg < -TOO_STEEP_PITCH_DEGREES;

            updateWarning();
        }
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        if(sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD) {
            mLowAccuracy = accuracy < SensorManager.SENSOR_STATUS_ACCURACY_HIGH;
            updateWarning();
        }
    }
};
```

7. Implement the `updateWarning` method, which displays a warning when magnetic interference is detected or when the pitch of the device approaches +/- 90 degrees and is prone to gimbal lock.

In `OrientationActivity.java`:

```
private void updateWarning() {
    String warning;

    if(mLowAccuracy) {
        warning = "Wear is detecting too much interference";
    } else if(mTooSteep) {
        warning = "The pitch value is approaching gimbal lock";
    }
}
```

```

    } else {
        warning = "";
    }

    mWarningText.setText(warning);
}

```

Now you can run this example and attempt to see how device orientations correspond to measurements of azimuth, pitch, and roll (see Figure 12-15).

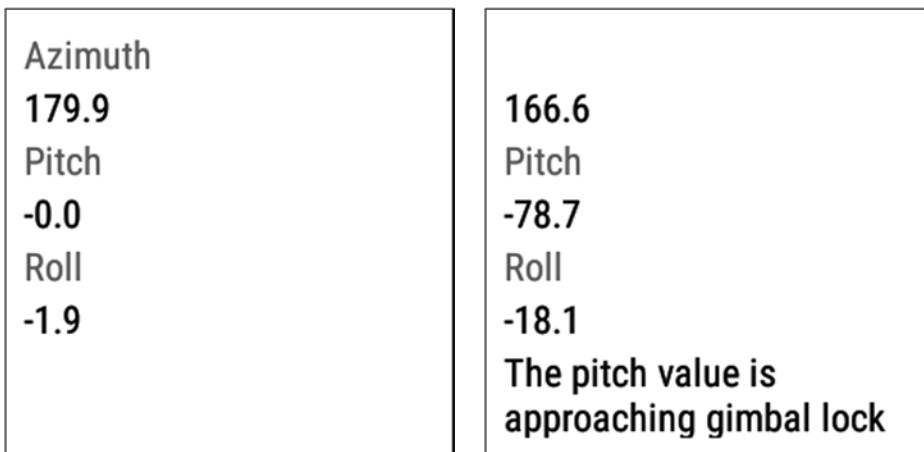


Figure 12-15. The output of `OrientationActivity` on an Android Wear device. On the left, the azimuth, pitch, and roll values show that the device is facing south. On the right, the pitch of the device is less than -70 degrees, which displays a warning message indicating that gimbal lock is approaching

Using Rotation Vector on Glass

This example will obtain updates from the rotation vector sensor on Glass, convert them to azimuth, pitch, and roll, and display them on the screen. Begin by creating an activity called `OrientationActivity.java` in the glass module.

1. Create a voice trigger with the “Demo orientation” keyword.

In res ► xml ► `orientation_trigger.xml`:

```

<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Demo orientation" />

```

2. Since we’re using an unlisted voice command, add the development permission.

In `AndroidManifest.xml`:

```

<uses-permission android:name="com.google.android.glass.permission.DEVELOPMENT" />

```

3. Configure the activity to be started in response to the voice command defined above.

In `AndroidManifest.xml`:

```
<activity
    android:name=".OrientationActivity">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>

    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/orientation_trigger" />
</activity>
```

4. Create a layout.

This layout contains `TextViews` that display the values of azimuth, pitch, and roll.

In `res > layout > activity_orientation.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:columnCount="2"
    android:paddingTop="40px"
    android:paddingLeft="40px"
    android:paddingRight="40px"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:text="Azimuth"
        style="@style/GlassTextSmall"
        android:textColor="#808080"
        android:layout_marginRight="20px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/azimuth"
        style="@style/GlassTextNormal"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:text="Pitch"
        style="@style/GlassTextSmall"
        android:textColor="#808080"
        android:layout_marginRight="20px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

```

<TextView
    android:id="@+id/pitch"
    style="@style/GlassTextNormal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:text="Roll"
    style="@style/GlassTextSmall"
    android:textColor="#808080"
    android:layout_marginRight="20px"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/roll"
    style="@style/GlassTextNormal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/warning"
    style="@style/GlassTextSmall"
    android:layout_columnSpan="2"
    android:layout_marginTop="15px"
    android:layout_gravity="center_horizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

</GridLayout>

```

5. Declare member variables.

Declare the member variables that are used throughout this example.

In `OrientationActivity.java`:

```

public class OrientationActivity extends Activity {
    private static final float TOO_STEEP_PITCH_DEGREES = 70.0f;
    private boolean mLowAccuracy, mTooSteep;
    private TextView mAzimuthText, mPitchText, mRollText, mWarningText;
    private SensorManager mSensorManager;
    private Sensor mRotationVectorSensor, mMagneticSensor;
    private float[] mRotationMatrix = new float[16];
    private float[] mOrientation = new float[3];
    ...
}

```

6. Initialize member variables.

Since this activity will not perform any actions when users swipe on the touchpad, wrap the layout with `TuggableView` to give users tugging feedback. `TuggableView` is available in the sample source code and was covered in Chapter 8.

In `OrientationActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    super.onCreate(savedInstanceState);
    View content = getLayoutInflater().inflate(R.layout.activity_orientation, null);
    setContentView(new TuggableView(this, content));

    mAzimuthText = (TextView) findViewById(R.id.azimuth);
    mPitchText = (TextView) findViewById(R.id.pitch);
    mRollText = (TextView) findViewById(R.id.roll);
    mWarningText = (TextView) findViewById(R.id.warning);

    mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    mRotationVectorSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR);
    mMagneticSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
}
```

7. Notify the user that tapping on the touchpad has no effect by playing the `Sounds.DISALLOWED` sound.

Recall that Glass automatically translates certain gestures into D-pad key events. Namely, tapping the touchpad triggers a key event with a key code of `KEYCODE_DPAD_CENTER`.

In `OrientationActivity.java`:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if(keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
        ((AudioManager) getSystemService(Context.AUDIO_SERVICE))
            .playSoundEffect(Sounds.DISALLOWED);
        return true;
    } else {
        return super.onKeyDown(keyCode, event);
    }
}
```

8. Register and unregister for sensor updates in `onResume` and `onPause`, respectively, to ensure that the app only receives updates when the activity is in the foreground.

In `OrientationActivity.java`:

```
@Override
protected void onResume() {
    super.onResume();
    registerForSensorUpdates();
}
```

```

private void registerForSensorUpdates() {
    mSensorManager.registerListener(mSensorEventListener, mRotationVectorSensor,
        SensorManager.SENSOR_DELAY_UI);

    // obtain accuracy updates from the magnetic field sensor since the rotation vector
    // sensor does not provide any
    mSensorManager.registerListener(mSensorEventListener, mMagneticSensor,
        SensorManager.SENSOR_DELAY_UI);
}

@Override
protected void onPause() {
    mSensorManager.unregisterListener(mSensorEventListener);
    super.onPause();
}

```

9. Sensor updates are passed to the `SensorEventListener` instance. In the `onSensorChanged` method, display the azimuth, pitch, and roll values in `TextViews`. Additionally, notify the user when the accuracy of the magnetic sensor decreases or when the pitch is too close to +/- 90 degrees.

In `OrientationActivity.java`:

```

private SensorEventListener mSensorEventListener = new SensorEventListener() {
    @Override
    public void onSensorChanged(SensorEvent event) {
        if(event.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR) {
            // convert rotation vector to azimuth, pitch, & roll
            SensorManager.getRotationMatrixFromVector(mRotationMatrix, event.values);

            // take into account Glass's coordinate system
            SensorManager.remapCoordinateSystem(mRotationMatrix, SensorManager.AXIS_X,
                SensorManager.AXIS_Z, mRotationMatrix);

            SensorManager.getOrientation(mRotationMatrix, mOrientation);
            float azimuthDeg = (float) Math.toDegrees(mOrientation[0]);
            float pitchDeg = (float) Math.toDegrees(mOrientation[1]);
            float rollDeg = (float) Math.toDegrees(mOrientation[2]);

            mAzimuthText.setText(String.format(Locale.US, "%.1f", azimuthDeg));
            mPitchText.setText(String.format(Locale.US, "%.1f", pitchDeg));
            mRollText.setText(String.format(Locale.US, "%.1f", rollDeg));

            mTooSteep = pitchDeg > TOO_STEEP_PITCH_DEGREES ||
                pitchDeg < -TOO_STEEP_PITCH_DEGREES;
            updateWarning();
        }
    }
}

```

```

@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
    mLowAccuracy = accuracy < SensorManager.SENSOR_STATUS_ACCURACY_HIGH;
    updateWarning();
}
};

```

10. Implement the `updateWarning` method, which displays a warning when magnetic interference is detected or when the pitch of the device approaches +/- 90 degrees and is prone to gimbal lock.

In `OrientationActivity.java`:

```

private void updateWarning() {
    String warning;
    if(mLowAccuracy) {
        warning = "Glass is detecting too much interference";
    } else if(mTooSteep) {
        warning = "The pitch value is approaching gimbal lock";
    } else {
        warning = "";
    }

    mWarningText.setText(warning);
}

```

When you run this example, you should see the device's azimuth, pitch, and roll as shown in Figure 12-16.

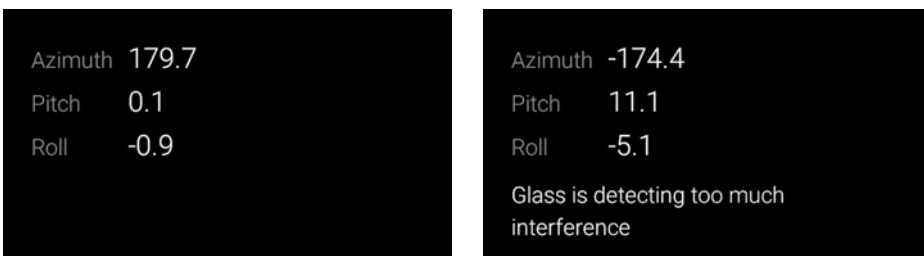


Figure 12-16. The output of `OrientationActivity` on Glass. On the left, the azimuth, pitch, and roll values show that the device is facing south. On the right, Glass is detecting magnetic interference and displays a warning

Building a Compass

We can use the rotation vector to display a compass that moves as the user rotates a wearable. The biggest difference between a compass and the previous examples is that the compass should indicate azimuth with respect to true north as opposed to magnetic north.

True North vs. Magnetic North

A compass points to magnetic north, which does not necessarily correspond to true north, which is also known as geographic north and is directed towards the north pole. The difference between true and magnetic north is known as declination and can be predicted for any given location at a particular time. The `GeomagneticField` class encapsulates this calculation.

Magnetic Interference

The rotation vector sensor is susceptible to magnetic interference, which may cause the azimuth to suddenly and drastically drift from its actual value. Magnetic interference is usually caused by having metallic or magnetic surfaces close to a device or by being inside buildings with thick walls. Most Android devices can detect the presence of magnetic interference and notify users that their compasses may not be accurate.

If magnetic interference is impacting a device's orientation measurement, try moving the device away from metallic or magnetic materials. Then, wave the device around in a figure eight motion to get the magnetometer to readjust to an environment without interference.

Building a Compass in Android Wear

This section demonstrates how to build a compass for Android Wear (see Figure 12-17). However, note that certain watches have issues measuring orientation. The Moto 360, in particular, does not always display azimuth with respect to true north. Also, the LG G watch does not call `onAccuracyChanged` and thus cannot display warnings when there is magnetic interference.

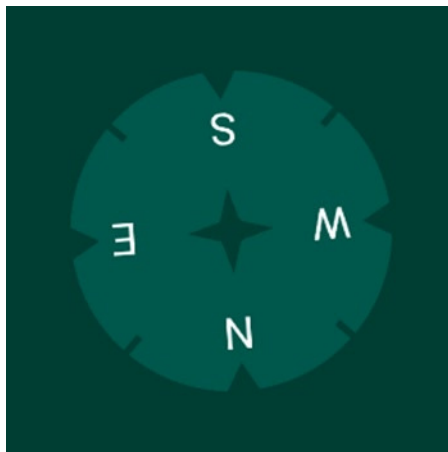


Figure 12-17. This example displays a compass that rotates as a user moves the watch

Create a View That Draws a Compass

CompassView draws a compass based on the device's current azimuth relative to true north.

1. Declare member variables.

Load the bitmap that contains the compass as a member variable to ensure that it's available when `onDraw` is called.

In `CompassView.java`:

```
public class CompassView extends View {
    private Bitmap mCompass;
    private float mAzimuth;
    private int mWidth, mHeight;
    private Rect mDestRect;
    private Paint mBitmapPaint;
    private int mOffsetX, mOffsetY;
    private int mBackgroundColor;
    ...
}
```

2. Initialize member variables.

The `mCompass` bitmap contains an image of a compass where north points towards the top. In the `onDraw` method, we'll draw this bitmap with the `mBitmapPaint` paint object, which specifies that it should be antialiased. The call to `setFilterBitmap(true)` is needed to properly antialias images.

In `CompassView.java`:

```
public CompassView(Context context) {
    super(context);
    init(context);
}

public CompassView(Context context, AttributeSet attrs) {
    super(context, attrs);
    init(context);
}

public CompassView(Context context, AttributeSet attrs, int defStyleAttr) {
    super(context, attrs, defStyleAttr);
    init(context);
}

private void init(Context context) {
    Resources res = context.getResources();
    mCompass = BitmapFactory.decodeResource(res, R.drawable.compass_wear);
    mBitmapPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mBitmapPaint.setFilterBitmap(true);
    mBackgroundColor = Color.parseColor("#004c3f");
}
```

3. Write a getter and a setter for azimuth.

If the azimuth changes value, invalidate the view to redraw the compass. This azimuth should be given with respect to true north.

In `CompassView.java`:

```
public float getAzimuth() {
    return mAzimuth;
}

public void setAzimuth(float azimuth) {
    if(mAzimuth != azimuth) {
        mAzimuth = azimuth;
        invalidate();
    }
}
```

4. Override the `onMeasure` method.

If `CompassView` is inflated with a width or height of `wrap_content`, attempt to take a size equal to the size of the compass bitmap. For more information on overriding `onMeasure`, see <http://developer.android.com/guide/topics/ui/custom-components.html>.

In `CompassView.java`:

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);

    int width = measureDimension(widthMeasureSpec, mCompass.getWidth());
    int height = measureDimension(heightMeasureSpec, mCompass.getHeight());
    int size = Math.min(width, height);
    setMeasuredDimension(size, size);
}

private int measureDimension(int measureSpec, int idealSize) {
    int result;
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);

    if(specMode == MeasureSpec.EXACTLY) {
        result = specSize;
    } else {
        result = idealSize;
        if(specMode == MeasureSpec.AT_MOST) {
            result = Math.min(result, specSize);
        }
    }

    return result;
}
```

5. Override the `onSizeChanged` method.

Initialize all variables that depend on the size of the view in this method.

In `CompassView.java`:

```
@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    super.onSizeChanged(w, h, oldw, oldh);
    mWidth = w - getPaddingLeft() - getPaddingRight();
    mHeight = h - getPaddingTop() - getPaddingBottom();
    mOffsetX = getPaddingLeft();
    mOffsetY = getPaddingTop();

    // scale bitmap to right size
    mDestRect = new Rect(0, 0, mWidth, mHeight);
}
```

6. Override the `onDraw` method.

The compass bitmap draws a compass that places north towards the top of the screen. In the `onDraw` method, rotate the compass bitmap rotated by `-azimuth` about the center of the view so that it points to true north.

In `CompassView.java`:

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    canvas.drawColor(mBackgroundColor);
    canvas.translate(mOffsetX, mOffsetY);
    canvas.rotate(-mAzimuth, mWidth/2f, mHeight/2f);
    canvas.drawBitmap(mCompass, null, mDestRect, mBitmapPaint);
}
```

Implementing `CompassActivity`

This activity obtains the device's orientation using the rotation vector sensor, calculates its azimuth relative to true north, and displays the compass using `CompassView`.

1. Create a layout.

This layout contains an instance of `CompassView`, which we implemented in the previous section, and a `TextView` that contains any warning or error messages.

In `res > layout > activity_compass.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:background="#004c3f"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```

<com.ocddevelopers.androidwearables.locationandorientation.CompassView
    android:id="@+id/compass"
    android:layout_margin="40px"
    android:layout_centerInParent="true"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>

<TextView
    android:id="@+id/warning"
    android:layout_centerInParent="true"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>

</RelativeLayout>

```

2. Declare `CompassActivity` as a launcher activity so it can be started with an app-provided voice action.

In `AndroidManifest.xml`:

```

<activity android:name=".CompassActivity"
    android:taskAffinity=
        "com.ocddevelopers.androidwearables.locationandorientation.WearCompass"
    android:label="Wearable Compass">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

```

3. Declare member variables.

In `CompassActivity.java`:

```

public class CompassActivity extends Activity {
    private static final float TOO_STEEP_PITCH_DEGREES = 70.0f;
    private static final int UPDATE_INTERVAL_MS = 60*1000; // 1 minute
    private static final int FASTEST_INTERVAL_MS = 500;
    public static final String TAG = "Compass";
    private boolean mLowAccuracy, mTooSteep;
    private SensorManager mSensorManager;
    private Sensor mRotationVectorSensor, mMagneticSensor;
    private CompassView mCompassView;
    private TextView mWarningText;
    private float[] mRotationMatrix = new float[16];
    private float[] mOrientation = new float[3];
    private GoogleApiClient mGoogleApiClient;
    private GeomagneticField mGeomagneticField;
    ...
}

```

4. Initialize member variables.

This example requires both location and orientation updates. Location updates are used to obtain the current declination so we can render the compass with respect to true north as opposed to magnetic north.

In `CompassActivity.java`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_compass);
    mCompassView = (CompassView) findViewById(R.id.compass);
    mWarningText = (TextView) findViewById(R.id.warning);

    mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    mRotationVectorSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR);
    mMagneticSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);

    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addApi(LocationServices.API)
        .addApi(Wearable.API)
        .addConnectionCallbacks(mConnectionCallbacks)
        .build();
}
```

5. Register and unregister for location and orientation updates.

Ensure that the app only receives location and orientation updates while in the foreground. Note that we don't register for location updates in `onResume`. Instead, we connect to `GoogleApiClient` from `onResume` and request location once the connection has been established, as we'll see shortly.

In `CompassActivity.java`:

```
@Override
protected void onResume() {
    super.onResume();
    registerForSensorUpdates();
    mGoogleApiClient.connect();
}

private void registerForSensorUpdates() {
    mSensorManager.registerListener(mSensorEventListener, mRotationVectorSensor,
        SensorManager.SENSOR_DELAY_UI);

    // obtain accuracy updates from the magnetic field sensor since the rotation vector
    // sensor does not provide any
    mSensorManager.registerListener(mSensorEventListener, mMagneticSensor,
        SensorManager.SENSOR_DELAY_UI);
}
```

```

@Override
protected void onPause() {
    mSensorManager.unregisterListener(mSensorEventListener);
    removeLocationUpdates();
    mGoogleApiClient.disconnect();
    super.onPause();
}

private void removeLocationUpdates() {
    if (mGoogleApiClient.isConnected()) {
        LocationServices.FusedLocationApi
            .removeLocationUpdates(mGoogleApiClient, mLocationListener);
    }
}

```

Update the Azimuth and Display Warnings

1. When a rotation vector update is received, calculate the current azimuth, which is given with respect to magnetic north, use it to calculate the azimuth relative to true north, and pass this value to `CompassView`. Additionally, notify the user when the accuracy of the magnetic sensor decreases or when the pitch is close to +/- 90 degrees.

In `CompassActivity.java`:

```

private SensorEventListener mSensorEventListener = new SensorEventListener() {
    @Override
    public void onSensorChanged(SensorEvent event) {
        if(event.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR) {
            // convert rotation vector to azimuth, pitch, & roll
            SensorManager.getRotationMatrixFromVector(mRotationMatrix, event.values);

            SensorManager.getOrientation(mRotationMatrix, mOrientation);
            float azimuthDeg = (float) Math.toDegrees(mOrientation[0]);
            float pitchDeg = (float) Math.toDegrees(mOrientation[1]);
            float rollDeg = (float) Math.toDegrees(mOrientation[2]);

            float trueNorthDeg = computeTrueNorth(azimuthDeg);
            mCompassView.setAzimuth(trueNorthDeg);

            mTooSteep = pitchDeg > TOO_STEEP_PITCH_DEGREES ||
                pitchDeg < -TOO_STEEP_PITCH_DEGREES;
            updateWarning();
        }
    }
}

```

```

@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
    mLowAccuracy = accuracy < SensorManager.SENSOR_STATUS_ACCURACY_HIGH;
    updateWarning();
}
};

```

2. Implement the `updateWarning` method, which displays a warning when magnetic interference is detected or when the pitch of the device approaches +/- 90 degrees and is prone to gimbal lock.

In `CompassActivity.java`:

```

private void updateWarning() {
    String warning;
    if(mLowAccuracy) {
        warning = "Wear is detecting too much interference";
    } else if(mTooSteep) {
        warning = "The pitch value is approaching gimbal lock";
    } else {
        warning = "";
    }

    mWarningText.setText(warning);
}

```

3. Implement the `computeTrueNorth` method, which calculate the azimuth relative to true north based on the azimuth relative to magnetic north and declination.

Recall that the declination is the difference between magnetic and true north. We obtain the declination using the `GeomagneticField` class.

In `CompassActivity.java`:

```

private float computeTrueNorth(float azimuthDeg) {
    if (mGeomagneticField != null) {
        return azimuthDeg + mGeomagneticField.getDeclination();
    } else {
        return azimuthDeg;
    }
}

```


Obtaining the Declination

As we saw in previous steps, the declination helps us find the direction of true north given the magnetic north. Calculate the declination with the `GeomagneticField` class based on the current location and time.

1. Implement the `requestLocationUpdate` method.

This method uses the Fused Location API to request for location updates, as we saw in the first example of this chapter.

In `CompassActivity.java`:

```
private void requestLocationUpdates() {
    LocationRequest locationRequest = LocationRequest.create()
        .setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY)
        .setInterval(UPDATE_INTERVAL_MS)
        .setFastestInterval(FATEST_INTERVAL_MS);

    LocationServices.FusedLocationApi
        .requestLocationUpdates(mGoogleApiClient, locationRequest, mLocationListener)
        .setResultCallback(new ResultCallback() {

            @Override
            public void onResult(Result result) {
                Status status = result.getStatus();
                if (!status.isSuccess()) {
                    Log.e(TAG, "Failed in requesting location updates, "
                        + "status code: "
                        + status.getStatusCode()
                        + ", message: "
                        + status.getStatusMessage());
                }
            }
        });
}
```

2. Any time the fused location provider calculates an updated location, use it to calculate the current declination with the `GeomagneticField` class.

In `CompassActivity.java`:

```
private LocationListener mLocationListener = new LocationListener() {
    @Override
    public void onLocationChanged(Location location) {
        updateDeclination(location);
    }
};
```

```
private void updateDeclination(Location location) {
    mGeomagneticField = new GeomagneticField((float)location.getLatitude(),
        (float)location.getLongitude(),
        (float)location.getAltitude(),
        location.getTime());
}
```

3. When the connection to `GoogleApiClient` is established, request location updates and attempt to calculate the declination best on the Fused Location API's most recent location.

In `CompassActivity.java`:

```
private GoogleApiClient.ConnectionCallbacks mConnectionCallbacks =
    new GoogleApiClient.ConnectionCallbacks() {
    @Override
    public void onConnected(Bundle bundle) {
        Location lastLocation =
            LocationServices.FusedLocationApi.getLastLocation(mGoogleApiClient);
        if(lastLocation != null) {
            updateDeclination(lastLocation);
        }

        requestLocationUpdates();
    }

    @Override
    public void onConnectionSuspended(int i) {
    }
};
```

Now you can run this example and test the compass on your watch. I recommend trying to modify the program to output the value of the current declination so you can see how much the difference between magnetic and true north is at your current location.

Building a Compass in Glass

This example demonstrates how to implement a compass on Glass and is very similar to the previous example, which implemented a compass for Android Wear. The compass indicates cardinal directions relative to true north (see Figure 12-18).

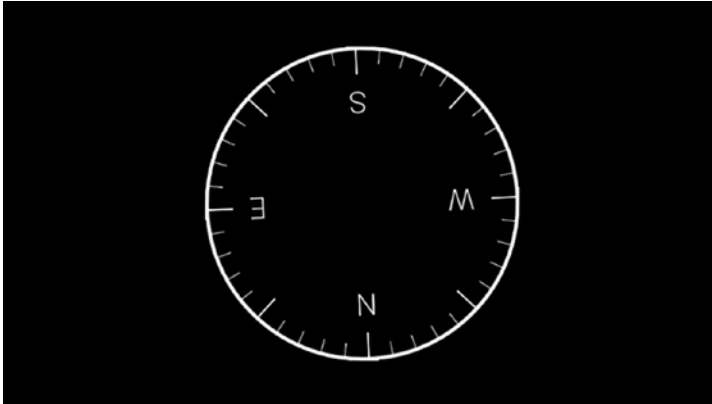


Figure 12-18. This example implements a compass on Glass that indicates cardinal directions relative to true north

Implementing CompassView

In the previous example, we implemented `CompassView` for Android Wear. In the current example, we implement a nearly identical version of `CompassView`, with only two minor changes to the `init` method (see lines in bold):

```
private void init(Context context) {
    Resources res = context.getResources();
    mCompass = BitmapFactory.decodeResource(res, R.drawable.compass_glass);
    mBitmapPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mBitmapPaint.setFilterBitmap(true);
    mBackgroundColor = Color.BLACK;
}
```

1. Create a layout.

The activity's layout contains an instance of `CompassView`, which we implemented in the previous section, and a `TextView` that contains any warning or error messages.

In `res > layout > activity_compass.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

<com.ocd.dev.androidwearables.chapter12.CompassView
    android:id="@+id/compass"
    android:layout_margin="40px"
    android:layout_centerInParent="true"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

```

<TextView
    android:id="@+id/warning"
    style="@style/GlassTextSmall"
    android:layout_centerHorizontal="true"
    android:layout_alignParentBottom="true"
    android:layout_marginBottom="40px"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

</RelativeLayout>

```

2. Create a voice trigger with the “Demo compass” keyword.

In res ► xml ► compass_trigger.xml:

```

<?xml version="1.0" encoding="utf-8"?>
<trigger keyword="Demo compass" />

```

3. Since we’re using an unlisted voice command, add the development permission.

In AndroidManifest.xml:

```

<uses-permission android:name="com.google.android.glass.permission.DEVELOPMENT" />

```

4. Declare CompassActivity as a launcher activity so it can be started with an app-provided voice action.

In AndroidManifest.xml:

```

<activity
    android:name=".CompassActivity">
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>

    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/compass_trigger" />
</activity>

```

5. Declare member variables.

In `CompassActivity.java`:

```
public class CompassActivity extends Activity {
    private static final float TOO_STEEP_PITCH_DEGREES = 70.0f;
    private static final int ARM_DISPLACEMENT_DEGREES = 6;
    private static final int LOCATION_UPDATE_INTERVAL_MS = 60000;
    private static final int LOCATION_DISTANCE_METERS = 2;
    private boolean mLowAccuracy, mTooSteep;
    private SensorManager mSensorManager;
    private Sensor mRotationVectorSensor, mMagneticSensor;
    private LocationManager mLocationManager;
    private CompassView mCompassView;
    private TextView mWarningText;
    private float[] mRotationMatrix = new float[16];
    private float[] mOrientation = new float[3];
    private GeomagneticField mGeomagneticField;
    ...
}
```

6. Initialize member variables.

This example requires both location and orientation updates. Location updates are used to obtain the current declination so we can render the compass with respect to true north as opposed to magnetic north.

@Override

```
protected void onCreate(Bundle savedInstanceState) {
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    super.onCreate(savedInstanceState);
    View content = getLayoutInflater().inflate(R.layout.activity_compass, null);
    setContentView(new TuggableView(this, content));

    mCompassView = (CompassView) content.findViewById(R.id.compass);
    mWarningText = (TextView) content.findViewById(R.id.warning);

    mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    mRotationVectorSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR);
    mMagneticSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
    mLocationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
}
```

7. Notify the user that tapping on the touchpad has no effect by playing the `Sounds.DISALLOWED` sound.

Recall that Glass automatically translates certain gestures into D-pad key events. Namely, tapping the touchpad triggers a key event with a key code of `KEYCODE_DPAD_CENTER`.

In `CompassActivity.java`:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if(keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
        ((AudioManager) getSystemService(Context.AUDIO_SERVICE))
            .playSoundEffect(Sounds.DISALLOWED);
        return true;
    } else {
        return super.onKeyDown(keyCode, event);
    }
}
```

8. Register and unregister for location and orientation updates in `onResume` and `onPause`, respectively.

In `CompassActivity.java`:

```
@Override
protected void onResume() {
    super.onResume();

    if(mGeomagneticField == null) {
        Location lastLocation = mLocationManager
            .getLastKnownLocation(LocationManager.PASSIVE_PROVIDER);
        if (lastLocation != null) {
            updateDeclination(lastLocation);
        }
    }

    registerForSensorUpdates();
    requestGpsUpdates();
}

private void registerForSensorUpdates() {
    mSensorManager.registerListener(mSensorEventListener, mRotationVectorSensor,
        SensorManager.SENSOR_DELAY_UI);

    // obtain accuracy updates from the magnetic field sensor since the rotation vector
    // sensor does not provide any
    mSensorManager.registerListener(mSensorEventListener, mMagneticSensor,
        SensorManager.SENSOR_DELAY_UI);
}

@Override
protected void onPause() {
    mSensorManager.unregisterListener(mSensorEventListener);
    removeGpsUpdates();
    super.onPause();
}

private void removeGpsUpdates() {
    mLocationManager.removeUpdates(mLocationListener);
}
```

9. Update the azimuth and display warnings.

When a rotation vector update is received, calculate the current azimuth, which is given with respect to magnetic north, use it to calculate the azimuth relative to true north, and pass this value to `CompassView`. Additionally, notify the user when the accuracy of the magnetic sensor decreases or when the pitch is close to +/- 90 degrees.

In `CompassActivity.java`:

```
private SensorEventListener mSensorEventListener = new SensorEventListener() {
    @Override
    public void onSensorChanged(SensorEvent event) {
        if(event.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR) {
            // convert rotation vector to azimuth, pitch, & roll
            SensorManager.getRotationMatrixFromVector(mRotationMatrix , event.values);

            // take into account Glass's coordinate system
            SensorManager.remapCoordinateSystem(mRotationMatrix, SensorManager.AXIS_X,
                SensorManager.AXIS_Z, mRotationMatrix);

            SensorManager.getOrientation(mRotationMatrix, mOrientation);
            float azimuthDeg = (float) Math.toDegrees(mOrientation[0]);
            float pitchDeg = (float) Math.toDegrees(mOrientation[1]);
            float rollDeg = (float) Math.toDegrees(mOrientation[2]);

            float trueNorthDeg = computeTrueNorth(azimuthDeg);
            // Glass's movable arm may be at an angle between 0 and 12 degrees. Assume
            // an average of 6 and draw the compass relative to the user's face.
            mCompassView.setAzimuth(trueNorthDeg - ARM_DISPLACEMENT_DEGREES);

            mTooSteep = pitchDeg > TOO_STEEP_PITCH_DEGREES ||
                pitchDeg < -TOO_STEEP_PITCH_DEGREES;
            updateWarning();
        }
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        mLowAccuracy = accuracy < SensorManager.SENSOR_STATUS_ACCURACY_HIGH;
        updateWarning();
    }
};
```

10. Implement the `computeTrueNorth` method, which calculate the azimuth relative to true north based on the azimuth relative to magnetic north and declination.

Recall that the declination is the difference between magnetic and true north. We obtain the declination using the `GeomagneticField` class.

In `CompassActivity.java`:

```
private float computeTrueNorth(float azimuthDeg) {
    if (mGeomagneticField != null) {
        return azimuthDeg + mGeomagneticField.getDeclination();
    } else {
        return azimuthDeg;
    }
}
```

11. Implement the `updateWarning` method, which displays a warning when magnetic interference is detected or when the pitch of the device approaches +/- 90 degrees and is prone to gimbal lock.

In `CompassActivity.java`:

```
private void updateWarning() {
    String warning;
    if(mLowAccuracy) {
        warning = "Glass is detecting too much interference";
    } else if(mTooSteep) {
        warning = "Keep Glass horizontal to use the compass";
    } else {
        warning = "";
    }
    mWarningText.setText(warning);
}
```

Obtain the Declination

As we saw in previous steps, the declination helps us find the direction of true north given the magnetic north. Calculate the declination with the `GeomagneticField` class based on the current location and time.

1. Implement the `requestGpsUpdates`, which requests updates from a list of location providers that obtain updates with a fine accuracy.

In `CompassActivity.java`:

```
private void requestGpsUpdates() {
    Criteria criteria = new Criteria();
    criteria.setAccuracy(Criteria.ACCURACY_FINE);
    criteria.setBearingRequired(false);
    criteria.setSpeedRequired(false);

    List<String> providers = mLocationManager.getProviders(criteria, true);

    for (String provider : providers) {
        mLocationManager.requestLocationUpdates(provider, LOCATION_UPDATE_INTERVAL_MS,
            LOCATION_DISTANCE_METERS, mLocationListener);
    }
}
```


2. When a location update is received, call the `updateDeclination` to calculate an updated declination.

In `CompassActivity.java`:

```
private LocationListener mLocationListener = new LocationListener() {
    @Override
    public void onLocationChanged(Location location) {
        updateDeclination(location);
    }

    @Override
    public void onStatusChanged(String provider, int status, Bundle extras) {
    }

    @Override
    public void onProviderEnabled(String provider) {
    }

    @Override
    public void onProviderDisabled(String provider) {
    }
};

private void updateDeclination(Location location) {
    mGeomagneticField = new GeomagneticField((float)location.getLatitude(),
        (float)location.getLongitude(),
        (float)location.getAltitude(),
        location.getTime());
}
```

At this point, you should be able to run this example on Glass.

Using Step Counter in Android Wear

Android supports many sensors in addition to the rotation vector sensor we've used so far. A very useful software sensor is called the step counter and serves as a pedometer. The step counter counts the number of steps a user takes. It consumes a low amount of power because it's implemented in hardware. After registering to receive updates from the step counter, it periodically passes the latest step count to a callback function. Certain Android mobile devices also support a sensor called the step detector which notifies the app every time a user takes a single step, but Android Wear devices only support the step counter. As of XE 22, Glass supports neither the step detector nor the step counter.

The step counter has a few caveats:

- it only counts steps while at least one application is registered for updates
- it returns the number of steps taken since last reboot

This example demonstrates how to use the step counter in Android Wear devices.

1. Create a layout.

The activity's layout contains a `TextView` that displays the current step count.

In `res > values > activity_pedometer.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/step_count"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:layout_centerInParent="true"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</RelativeLayout>
```

2. Declare `PedometerActivity` as a launcher activity so it can be started with an app-provided voice action.

In `AndroidManifest.xml`:

```
<activity android:name=".PedometerActivity"
    android:label="Wearable Pedometer">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3. Declare and initialize member variables.

In `PedometerActivity.java`:

```
public class PedometerActivity extends Activity {
    private TextView mStepCountText;
    private SensorManager mSensorManager;
    private Sensor mStepCounterSensor;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_pedometer);
    }
}
```

```

    mStepCountText = (TextView) findViewById(R.id.step_count);
    mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    mStepCounterSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_STEP_COUNTER);
}
...

```

4. Register and unregister to receive updates from the step counter.

Register and unregister the step counter in `onResume` and `onPause` to ensure that updates are only received while the app is in the foreground.

In `PedometerActivity.java`:

```

@Override
protected void onResume() {
    super.onResume();
    mSensorManager.registerListener(mSensorEventListener, mStepCounterSensor,
        SensorManager.SENSOR_DELAY_NORMAL);
}

@Override
protected void onPause() {
    mSensorManager.unregisterListener(mSensorEventListener);
    super.onPause();
}

```

5. Display the most recent step count.

Display the step count, which is returned in `event.values[0]`, on a `TextView`.

In `PedometerActivity.java`:

```

private SensorEventListener mSensorEventListener = new SensorEventListener() {
    @Override
    public void onSensorChanged(SensorEvent event) {
        if(event.sensor.getType() == Sensor.TYPE_STEP_COUNTER) {
            int steps = Math.round(event.values[0]);
            mStepCountText.setText(Integer.toString(steps));
        }
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
    }
};

```

Now you can run the example and see the number of steps since the last reboot as measured by the watch.

Summary

In this chapter, we learned about the coordinate systems that Android uses for both location and orientation updates. We also learned how to access location and orientation on both Android Wear and Glass. We then implemented a compass to highlight the difference between true and magnetic north, and we concluded by implementing a step counter on Android Wear and noted that this sensor is not available on Glass.

Index

A

Accelerometers, [443](#)
ACTION_CHANGE_UNITS action, [334–335](#)
ActionFragment

- Intent actionIntent, [151](#)
- int iconId, [151](#)
- member variables, [151](#)
- newInstance method, [152](#)
- OnLayoutInflatedListener listener, [153](#)
- static method, [151](#)
- String label, [151](#)
- WatchViewStub, [152](#)

addNextIntentMethod, [70](#)
ALERT layout

- DialogTheme style, [311](#)
- generic motion events, [312](#)
- GestureDetector.BaseListener, [312](#)
- GlassAlertDialog, [313](#)
- member variable, [311](#)
- onClickListener, [312](#)
- voice command, [310](#)

Altitude, [423](#)
Android Wear, [95](#)

- compass design in, [456–457](#)
- Hello Wear activity
 - implementation, [101](#)
 - taskAffinity, [101](#)
 - TextView, [101](#)
- project creation, [96](#)
- rotation vector sensor in, [444–445](#)
- step counter in, [472](#)
- timer app, [114](#)
 - CommandReceiver, [116, 121](#)
 - setUsesChronometer, [116](#)
 - TimerActivity, [116, 126](#)
 - TimerCompletedActivity, [115, 124](#)
 - timer notification, [115](#)
 - TimerUtil, [115, 117](#)

- voice actions, [98](#)
 - app-provided voice actions, [99](#)
 - system-provided
 - voice actions, [99](#)
- wearable UI library
 - (see [WearableListView](#))

Android Wear, location updates on, [424](#)

- FusedLocationProvider
 - Interval and FastestInterval, [430](#)
 - LocationRequest Priority, [430](#)
- node and location
 - updates registering, [427](#)
 - requesting/removing updates, [430](#)

AUTHOR layout, [308–309](#)
Azimuth and display warnings, [462](#)
Azimuth, Pitch, and Roll, [440](#)

B

BigPictureStyle Notification, [38, 61](#)
BigTextStyle notification

- chat messages, [37](#)
- expandable card, [59](#)
- implementation, [37–38](#)
- menu item, [61](#)
- static card, [60](#)

BigTextStyle page, [77](#)
BoxInsetLayout

- layout_box parameter, [159](#)
- round watch, [159](#)
- running stats
 - location/GPS, [160](#)
 - MIN_SPEED_MPH and MAX_SPEED_MPH, [161–163](#)
 - RunCallback, [161](#)
 - SimulatedRunStatsReceiver
 - class, [160](#)
- RunStatsActivity, [164](#)
- square watch, [159](#)

C

callback method, 326

Camera API, 391

activity layout, 396

CameraUtils, 392

getCameraInstance, 392

getOutputMediaFile, 392

getOutputPictureFile, 393

scanFile method, 393

ImageView, 390

layout creation, 384

LiveCards, 402

CameraLiveCardService, 406

CameraPreview, 402

menu creation, 404

renderingPaused method, 402, 404

surfaceChanged method, 403

surfaceCreated method, 402

surfaceDestroyed method, 403

SurfaceHolder, 402

onActivityResult method, 388

onCreate method, 397

onOptionsMenuClosed, 391

options menu, 386

processPictureWhenReady method, 389

recording videos, 383

startRecordVideoIntent methods, 388

SurfaceHolder, 394

callback methods, 394

camera parameters, 394

implementation, 394

taking pictures, 382

time-lapse, 407

activity, 408

initSoundPool method, 409

invalidateOptionsMenu, 412

onCreate method, 409

onCreateOptionsMenu, 411

options menu, 410

playSound method, 410

prepareVideoRecorder method, 413

recording, 412

Record time-lapse, 410

releaseMediaRecorder method, 414

Stop time-lapse, 410

toggleRecording method, 412

voice command, 384

Canvas class, 328

CAPTION layout, 307

CardBuilder class, 295

actions, 295

ALERT layout

DialogTheme style, 311

generic motion events, 312

GestureDetector.BaseListener, 312

GlassAlertDialog, 313

member variable, 311

onClickListener, 312

voice command, 310

AUTHOR layout, 308–309

CAPTION layout, 307

CardScrollAdapter

creation, 298–299

getItem method, 297

getItemViewType method, 298

getPosition method, 298

getViewTypeCount method, 297

member variable, 296

COLUMNS_FIXED layout, 306–307

COLUMNS layout, 304

Digital Speedometer

(see CardBuilder class)

LiveCards (see LiveCards)

MENU layout, 309

slider

demoGracePeriodSlider

method, 346–347

demoIndeterminateSlider

method, 345

determinate progress, 341–342

grace period, 342

indeterminate progress, 341

member variables, 343

options menu, 344

scroller sliders, 342

voice trigger, 342

TEXT_FIXED layout, 303–304

TEXT layout, 300

addImage method, 301–302

background image, 302

ellipsizing text, 300

initCards method, 303

TITLE layout, 308

Voice Command

declaration, 296

Card Expansion property, 131

CardFragment

- Card Expansion property, 131
- Card Gravity Property, 130–131
- card layout, 130
- Expansion Direction property, 131
- Expansion Factor property, 132

Card Gravity property, 130–131

CircledImageView, 143

COLUMNS_FIXED layout, 306–307

COLUMNS layout, 304

Compass, 455

- design in Android Wear, 456–457
- design in Glass, 465
 - CompassView implementation, 466
 - declination, 471
- magnetic interference, 456
- true north vs. magnetic north, 456

CompassActivity implementation, 459

- azimuth and display warnings, 462
- declination, 464

Compass app, 419

ConfirmationActivity, 147–148

ConfirmationDemoActivity

- action buttons
 - fragment creation, 151
 - round watche layout creation, 150
 - square watche layout creation, 149
 - WatchViewStub, 151
- DelayedConfirmationActivity, 154–156
- FragmentGridPagerAdapter, 158
- main screens, 148
- makeConfirmation
 - ActivityIntent method, 157–158
- member variables, 157
- window overscan, 156

Coordinate frames, 438

Earth/world, 439

inertial/device, 438

Custom notification, 167

- characteristics of, 169
- constants, 168
- features, 45
- implementation
 - expanded view, 47–48
 - getActivityPendingIntent method, 47

- getMediaCommandPendingIntent
 - method, 46
- onCustomNotificationButtonClick
 - method, 46
- MediaStyle notification
 - (see MediaStyle notification)
- PendingIntent, 168
- running stats, 167
- RunStatsActivity class, 168
- start, 171
- Stop Run action, 170

D

Data API, 188

- DataEventBuffer, 204
- DataListener, 202, 204
- decrement and
 - increment buttons, 205
- definition, 201
- getDataItems method, 203
- path, 201
- payload, 201
- PutDataMapRequest class, 201
- retrieving assets, 211
- runOnUiThread, 203–204
- sending assets, 207
- updateCountData, 205

DataEventBuffer, 204

Datum, 420

DelayedConfirmationActivity, 154–156

DelayedConfirmationView

- alarm icon, 145
- ConfirmationActivity, 147–148
- ConfirmationDemoActivity
 - (see ConfirmationDemoActivity)
- onTimerFinished, 145
- WatchViewStub, 146
- Window Overscan, 146
- XML layout, 144–145

demoDeterminateSlider method, 345

demoGracePeriodSlider method, 346

demoIndeterminateSlider method, 345

Digital Speedometer

- aspects of, 330
- DigitalSpeedMenuActivity, 332

Digital Speedometer (*cont.*)

- DigitalSpeedRenderer, 337
- DigitalSpeedService, 335
- DigitalSpeedView, 331
- RenderThread, 339

draw method, 327

E

Earth as ellipsoid, 419

Earth/world coordinate frame, 439

Equatorial plane, 420–421

Expansion Direction property, 131

Expansion Factor property, 132

F

fetchLocalNode, 186

fetchRemoteNode, 186–187

FLAG_ACTIVITY_NEW_TASK flag, 317

flag Intent.FLAG_ACTIVITY_SINGLE_TOP, 74

formatMessage method, 77, 80

FusedLocationProvider

- Interval and FastestInterval, 430

- LocationRequest Priority, 430

G, H

Geodetic latitude and longitude, 420

- equatorial plane, 420–421

- latitude definition, 422

- longitude definition, 421–422

- meridians of longitude, 421

- prime meridian, 421

Gestures, touchpad

- GestureDetector, 354

- BaseListener, 356

- CardBuilder, 356

- FingerListener, 358

- GestureActivity, 355

- onCreate method, 356

- onGenericMotionEvent, 356

- ScrollListener, 358

- head gestures, 366

- detectNod method, 370

- gyroscopic sensor, 367, 373

- HeadGestureActivity, 371

- HeadGestureDetector, 369, 373

- HeadGestureListener, 369

- NodDetector, 368–369

- onCreate method, 372

- onSensorChanged method, 370

- triggerNod method, 371

- updateText, 373

- inertial scrolling, 364

- AnimatorUpdateListener, 365

- DecelerateInterpolator, 365

- onScroll method, 365

- ValueAnimator, 364–365

- MotionEventActivity, 350

- MotionEvent object, 349

- onGenericMotionEvent, 349, 354

- onTouchEvent, 349

- getActivityPendingIntent method, 47

- getBackgroundForPage method, 134

- getDataItems method, 203

- getItem method, 297

- getItemViewType method, 298

- getMediaCommandPendingIntent method, 46

- getPicture method, 210

- getPosition method, 298

- getScaledLargeIconFrom

- Resource method, 87

- getViewTypeCount method, 297

Glass

- cards, 12

- compass design in, 465

- CompassView implementation, 466

- declination, 471

- explorers program, 13

- Google IO, 10

- lifelogging and photography

- applications, 13

- location updates on, 432, 435

- MyGlass app, 12

- notification sync, 13

- overview, 9

- rotation vector sensor, 445, 450

- timeline, 11

- voice commands, 12

- Glass Development Kit (GDK), 275

- CardScrollView, 285

- definition, 275

playSoundEffect method, 285
uses, 276

Glassware, 275. *See also* Glass
code implementation, 280
combined applications, 279

GDK
CardScrollView, 285
definition, 275
playSoundEffect method, 285
uses, 276

immersions, 277, 283

Mirror API
definition, 276
timeline items, 276
uses, 276

ongoing tasks, 278

ScrollingCardsActivity, 286

TuggableView, 290

voice commands, 282

GoogleApiClient
asynchronous connection
build.gradle file, 175
ConnectionCallbacks, 175
manifest, 175
member variable
declaration, 175
mResolvingError flag, 178
onActivityResult method, 177
onConnected method, 176
OnConnectionFailedListener, 176
standard launcher activity
declaration, 175
UI thread, 174
using minimal version, 179
synchronous connection, 180

Google Glass. *See* Glass

GPS provider, 423

GridViewPager, 132
creation, 141
fixed movement paging, 140

FragmentManager
Adapter class, 134

getBackground
ForPage method, 134

grid of, 133

VocabularyAdapter, 138

vocabulary words, 135–136

Gyroscopes, 443

I, J

InboxStyle notification, 40, 63

Inertial/device coordinate frame, 438

initCards method, 303

Interruptive priority, 34, 36

Interval and FastestInterval, 430

K

Kalman filter, 443

L

Latitude, 422

LiveCards, 402
CameraLiveCardService, 406
CameraPreview, 402
high-frequency rendering, 321
RenderThread, 329
SpeedMenuActivity, 324
SpeedRenderer, 325
SpeedService, 322
usage, 322

low-frequency rendering, 316
BroadcastReceiver, 319
layout creation, 317
member variables, 317
onBind method, 319
setAction method, 317
updateRemoteViews, 318
usage, 316
voice trigger, 316

menu creation, 319, 404
options menu, 315
publish mode, 315
renderingPaused method, 402, 404
surfaceChanged method, 403
surfaceCreated method, 402
surfaceDestroyed method, 403
SurfaceHolder, 402
update, 315

loadBitmapFromAsset method, 212

Location and orientation
altitude, 423
compass, 455
design in Android Wear, 456
magnetic interference, 456
true north vs. magnetic north, 456

Location and orientation (*cont.*)

- CompassActivity implementation, 459
 - azimuth and display warnings, 462
 - declination, 464
- compass app, 419
- compass design in Glass, 465
 - CompassView implementation, 466
 - declination, 471
- Earth as ellipsoid, 419
- geodetic latitude and longitude, 420
 - equatorial plane, 420–421
 - latitude definition, 422
 - longitude definition, 421–422
 - meridians of longitude, 421
 - prime meridian, 421
- location providers, 423
 - Google Play Services, 424
 - GPS provider, 423
 - network provider, 423
 - passive provider, 424
 - selection of, 424
- location updates on Android Wear, 424
 - FusedLocationProvider, 429
 - node and location updates
 - registering, 427
 - requesting/removing updates, 430
 - location updates on Glass, 432, 435
 - navigation app, 419
 - step counter in Android Wear, 472
- Location providers, 423
 - Google Play Services, 424
 - GPS provider, 423
 - network provider, 423
 - passive provider, 424
 - selection of, 424
- LocationRequest Priority, 430
- Longitude, 421–422

M

- Magnetic interference, 456
- Magnetometers, 443
- makeConfirmationActivityIntent method, 157
- MediaStyle notification
 - compatibility library, 49
 - creation, 52–53
 - declare and initialize member variables, 50
 - Music Playback, 51

- onMediaStyleNotification
 - ButtonClick method, 54
 - play and pause actions, 53–54
- MENU layout, 309
- Meridians of longitude, 421
- Message API, 188
 - handheld activity, 190
 - receiving, 197–198
 - request response paradigm, 198
 - sending, 196–197
 - usage, 188
 - wearable activity, 193
- MessageListener, 200
- Moto 360 smartwatch, 443

N

- Navigation app, 419
- Network provider, 423
- NodeListener, 427–428
- Notifications, 27
 - in Android 5.0
 - category, 30
 - interruption filter, 29
 - light background, 29
 - compatibility, 36
 - custom notifications
 - (see Custom notifications)
 - HandheldNotifications project
 - activity declaration, 21
 - activity implementation, 21–22
 - layout creation, 20
 - MainActivity, 19
 - heads-up
 - implementation, 35–36
 - interruptive priority, 34
 - lock screen, 30
 - Contents hidden, 33
 - private notification, 32
 - public notification, 32
 - secret notification, 32
 - sensitive notification content, 32
 - settings, 31
 - NotificationCompat's setColor method, 36
 - rich notifications (see Rich notifications)
 - standard notifications
 - (see Standard notifications)
- Notification's priority, 27

O

- onActivityResult method, 177
- onBackgroundOnlyImageClick method, 89
- onBind method, 319
- onBuildTaskStackContentIntentClick method, 70
- onConnected method, 176, 427
- OnConnectionFailedListener, 176
- onContentActionClick method, 91
- onCustomNotification
 - ButtonClick method, 46
- OnLayoutInflatedListener listener, 153
- onMediaStyleNotification
 - ButtonClick method, 54
- onMessageReceived method, 199
- onNotificationActionsButtonClick method, 43
- onPagingClick method, 78
- onPrivateNotificationWithPublic
 - VersionClick method, 33
- onPublicNotificationClick method, 32
- onStackingClick method, 82
- onStartCommand, 335
- onStartCommand method, 334
- onTimerFinished method, 145
- onVoiceReplyClick method, 86
- onWearableOnlyActionsClick method, 74
- Orientation, 437
 - Azimuth, Pitch, and Roll, 438, 440
 - coordinate frames, 438
 - Earth/world, 439
 - inertial/device, 438
 - measurement, 443
 - accelerometers, 443
 - gyroscopes, 443
 - Kalman filter, 443
 - magnetometers, 443
 - Moto 360 smartwatch, 443
 - rotation vector sensor, 444–445, 450
 - rotation matrices and quaternions, 442
- Orientation. See Location and orientation

P, Q

- Passive provider, 424
- Peek cards, 221
 - ambient mode, 225
 - card peek mode, 224
 - interactive mode, 226

- Positive rotation, 441
- Prime meridian, 421
- PRIORITY_BALANCED_POWER_ACCURACY, 430
- PRIORITY_HIGH_ACCURACY, 430
- PRIORITY_LOW_POWER, 430
- PRIORITY_NO_POWER, 430
- Private notification, 32
- Public notification, 32
- publishCard method, 322

R

- Reference ellipsoid, 420
- renderingPaused method, 326
- Rich notifications
 - actions, 42
 - MediaCommandService service, 43
 - onNotificationActionsButtonClick method, 43
 - pause & next, 43
 - recommended size, large icon, 44
 - text-to-speech, 43
 - BigPictureStyle Notification, 38
 - BigTextStyle notification
 - chat messages, 37
 - implementation, 37
 - InboxStyle notification, 40
 - normal and expanded state, 36
- Right-hand rule, 441
- Rotation matrices and quaternions, 442
- Rotation vector sensor, 444
 - in Android Wear, 444
 - in Glass, 445
 - implementation in
 - Android Wear, 445
 - usage on Glass, 450
- RunStatsActivity, 164
- runStatsReceived method, 166

S

- Secret notification, 32
- setTotalTimeMs method, 145
- Stacking notifications
 - context stream, 80
 - formatMessage method, 80
 - InboxStyle, 79
 - InboxStyle line, 81

Stacking notifications (*cont.*)

- onStackingClick method, 82
- TextAppearanceSpan, 81
- WearableExtender, 82

Standard notifications, 58

- content intent, 23, 59
- flight status notification, 22
- implementation, 24
- PendingIntents, 23
- requirements, 22
- single top flag, 24
- update, 26

StartConfirm Wearable Counter, 189, 198

static method, 139

Step counter, 472

T

Take pictures and record videos.

See Camera API

TEXT_FIXED layout, 303–304

TEXT layout, 300

- addImage method, 301–302
- background image, 302
- ellipsizing text, 300
- initCards method, 303

Time-lapse, 407

- activity, 408
- initSoundPool method, 409
- invalidateOptionsMenu, 412
- onCreate method, 409
- options menu, 410
- playSound method, 410
- prepareVideoRecorder method, 413
- recording, 412
- Record time-lapse, 411
- releaseMediaRecorder method, 414
- Stop time-lapse, 411
- toggleRecording method, 412

Timer app, 115

- CommandReceiver, 116, 121
- setUsesChronometer, 116
- TimerActivity, 116, 126
- TimerCompletedActivity, 115, 124
- timer notification, 115
- TimerUtil, 115, 117

TITLE layout, 308

U

unpublishCard method, 323

updateConnectionStatus method, 186

updateCountData method, 205

updateCountFromDataItem method, 203

updateRendering method, 327

updateText method, 332

updateWarning method, 428–429

V

Voice commands, 282

Voice recognition, 374

- contextual voice commands, 377
- displaySpeechRecognizer, 376
- onActivityResult method, 376
- onCreatePanelMenu, 378
- voice commands, 375

W, X, Y, Z

Watch faces, 216

ambient mode, 216–217

boilerplate code, 230

burn-in protection, 219

configuration, 222

- background visibility, 227
- hotword indicator gravity, 227
- setAmbientPeekMode method, 225
- setCardPeekMode method, 224
- setPeekOpacityMode, 226
- status bar gravity, 223
- system UI time, 229
- view protection, 228

Convergence, 245

- background, 248
- handheld configuration, 247
- interactive mode, 246
- wearable configuration, 246

creation, 230

data API, 252

- continuous sweep, 267
- ConvergenceConfigActivity, 252, 259
- ConvergenceUtil class, 253
- ConvergenceWatchFaceService, 252
- ConvergenceWearable
 - ConfigActivity, 252, 256

- ConvergenceWearable
 - ListenerService, 252, 262
 - paints and bitmaps, 265
- declaration, 231
- Engine class, 233
 - burn-in protection, 233
 - constants and member variables, 233
 - detection, 236
 - low-bit ambient mode, 233
 - mTime instance, 236
 - onDestroy() method, 236
 - onVisibilityChanged, 235
 - paint objects, 238
 - peek card, 233
 - square/round, 234
 - tick event, 234
 - updateColors method, 238
 - updateTimer method, 235
- graphic design program, 239
- hotword indicator, 221
- interactive mode, 216
- interruption filter, 221
- LCD screens, 218
- low-bit ambient mode, 218
- Moto 218, 220, 360
- OLED screen, 218
- onDraw method, 240
- peek card, 221
- square and round screens, 220
- status bar, 221
- WatchFaceBitmapHolder, 249
- wear module, 232
- WatchViewStub, 146
- Wearable data layer API, 173
 - data API (see Data API)
 - GoogleApiClient (see GoogleApiClient)
 - message API, 188
 - handheld activity, 190
 - receiving, 197–198
 - request response paradigm, 198
 - sending, 196–197
 - usage, 188
 - wearable activity, 193
- PendingResult, 181
- types, 173
 - using node API
 - ConnectionCallbacks, 185
 - fetchLocalNode, 186
 - fetchRemoteNode, 187
 - layout creation, 182–183
 - member variables, 184
 - NodeListener
 - implementation, 185
 - standard launcher
 - activity declaration, 184
 - updateConnection
 - Status method, 186
 - usage, 182
- WearableListenerService, 197
- WearableListView, 103
 - animation, 107
 - definition, 102
 - float values, 110
 - Object target, 110
 - onClick method, 107
 - RecyclerView, 103
 - SimpleWearableListViewActivity, 105
 - String propertyName, 110
 - view holder pattern, 103
 - WearableAdapter, 112
 - WearableListItemLayout, 107
 - WearableListView.Adapter, 104
- Wearable notifications
 - actions, 64, 66
 - ActionFeedbackActivity, 72–75
 - creation, 71
 - MainActivity.java, 73
 - NotificationCompat.Action
 - constructor, 74
 - PendingIntent, 71
 - using arbitrary icons, 74
 - WearableExtender, 75
 - background only notifications, 88
 - BigPictureStyle notifications, 61
 - BigTextStyle notifications
 - expandable card, 59
 - menu item, 61
 - static card, 60
 - content action, 90–92
 - InboxStyle notification, 63
 - MainActivity, 56–57

Wearable notifications (*cont.*)

- pages
 - BigTextStyle page, 77
 - CharSequence creation, 77
 - conversation history, 75–76
 - formatMessage method, 77
 - onPagingClick method, 78
 - stacking notifications (see Stacking notifications)
 - standard notifications (see Standard notifications)
 - voice input notification, 83–84
 - ChatDetailActivity, 85
 - getScaledLargeIconFromResource method, 87
 - onVoiceReplyClick method, 86
 - RemoteInput, 86
 - Wielding TaskStackBuilder
 - activities, 67
 - ChatDetailActivity, 68
 - definition, 66
 - getConversationPendingIntent, 70
 - meta-data tag, 69
 - PendingIntent, 67, 70
- Wearables devices, 3
- Android Wear, 8
 - benefits, 5
 - collaborator, 5
 - context stream, 6
 - cue card, 7
 - overview, 5
 - sleep trackers, 13
 - BioGlass, 4
 - definition, 3
 - design matters, 14
 - Glass
 - cards, 12
 - explorers program, 13
 - Google IO, 10
 - lifelogs and
 - photography applications, 13
 - MyGlass app, 12
 - notification sync, 13
 - overview, 9
 - timeline, 11
 - voice commands, 12
 - LynxFit Glassware, 4
 - user experience, 4
 - WGS-84 datum, 420
 - Wielding TaskStackBuilder
 - activities, 67
 - ChatDetailActivity, 68
 - definition, 66
 - getConversationPendingIntent, 70
 - meta-data tag, 69
 - PendingIntent, 67, 70

Beginning Android Wearables



Andres Calvo

Apress®

Beginning Android Wearables

Copyright © 2015 by Andres Calvo

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4842-0518-1

ISBN-13 (electronic): 978-1-4842-0517-4

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The images of the Android Robot (01/Android Robot) are reproduced from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License. Android and all Android and Google-based marks are trademarks or registered trademarks of Google Inc. in the United States and other countries. Apress Media LLC is not affiliated with Google Inc., and this book was written without endorsement from Google Inc.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr

Acquisitions Editor: Steve Anglin

Development Editor: Jane Hosie-Bounar

Technical Reviewer: Jeff Tang

Editorial Board: Steve Anglin, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Michelle Lowman,

James Markham, Susan McDermott, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick,

Ben Renow-Clarke, Gwenan Spearing, Steve Weiss

Coordinating Editor: Mark Powers

Copy Editor: Lindsay Beaton

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

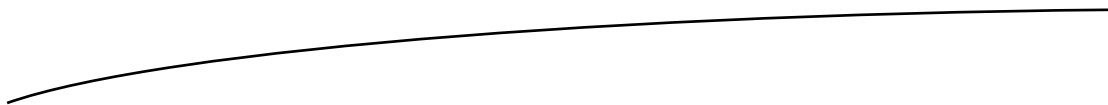
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales-eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com/9781484205181. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.



*To my mother and father, for showing me the
way to where I am today and to where I'll be tomorrow.*

Contents

About the Author	xix
About the Technical Reviewer	xxi
Acknowledgments	xxiii
■ Part I: Introduction	1
■ Chapter 1: Introducing Android Wearables	3
Wearables and Contextual Awareness	3
Wearables and User Experience	4
Why Android Wearables?	5
What Can Android Wear Do?	5
The Context Stream	6
The Cue Card	7
The Android Wear App	8
What Can Glass Do?	9
Understanding Glass Lingo	10
Google Glass vs. Glass	10
Glassware	10
Glass Explorers	10

Cards & Timeline.....	11
Voice Commands.....	12
The MyGlass App.....	12
The End of the Glass Explorers Program.....	13
Should I Develop for Glass or Android Wear?.....	13
Design Matters.....	14
Reading This Book.....	14
Summary.....	15
■ Part II: Notifications.....	17
■ Chapter 2: Reviewing Notifications for Android.....	19
The Example App.....	19
Standard Notifications.....	22
Always Use a Content Intent.....	23
PendingIntents.....	23
A Single Top Activity.....	24
Implementation.....	24
Updating Notifications.....	26
Implementation.....	26
Notification Priority.....	27
Notification Alerts.....	27
Notifications in Android 5.0.....	29
Dark Content on a Light Background.....	29
The Interruption Filter.....	29
Notification Category.....	30
Lock Screen Notifications.....	30
Heads-up Notifications.....	34
Changing a Notification's Color.....	36
Compatibility.....	36

Rich Notifications	36
BigTextStyle Notification.....	37
BigPictureStyle Notification.....	38
InboxStyle Notification.....	40
Notification Actions	42
Implementation.....	43
Custom Notifications	45
Implementation.....	46
MediaStyle Notification	49
Implementing MediaStyleService.....	49
Handle the Play and Pause Actions	53
Implementing MainActivity.....	54
Summary.....	54
■ Chapter 3: Customizing Notifications for Wearables.....	55
Getting Started	55
The Example App.....	56
Handheld Notifications on Wearables.....	58
Standard Notifications	58
BigTextStyle Notifications.....	59
BigPictureStyle Notifications	61
InboxStyle Notifications.....	63
Notification Actions.....	64
Customizing Notifications for Wearables.....	66
Wielding TaskStackBuilder	66
Wearable-Only Actions	71
Notification Pages.....	75
Stacking Notifications.....	79
Voice Input Notification.....	83
Background Only Notifications	87
Content Action	90
Summary.....	92

- Part III: Android Wear 93**
- Chapter 4: Running Apps Directly on Android Wear 95**
 - The Android SDK in Android Wear 95
 - Creating a New Project..... 96
 - Starting Wear Apps..... 98
 - App-Provided Voice Actions..... 99
 - System-Provided Voice Actions 99
 - The Example App..... 100
 - Example #1: Our First Wearable App 100
 - The Wearable UI Library 102
 - Our First Wearable UI View: WearableListView 102
 - WearableListView and RecyclerView..... 103
 - The ViewHolder Pattern 103
 - Example #2: Implementing a List of Strings 103
 - Example #3: Implementing an Animated WearableListView 107
 - Implementing AnimatedListViewActivity 113
 - Example #4: Creating a Timer App 114
 - Overview of the Architecture 115
 - Implementing TimerUtil 117
 - Implementing CommandReceiver 121
 - Implementing TimerCompletedActivity..... 124
 - Implementing TimerActivity..... 126
 - Summary..... 128
- Chapter 5: Android Wear User Interface Essentials 129**
 - Using CardFragment..... 130
 - The Card Gravity Property..... 130
 - The Card Expansion Property 131
 - The Expansion Direction Property 131
 - The Expansion Factor Property..... 132
 - Using GridViewPager..... 132

Example App #1: Vocab Builder	135
Representing Vocabulary Words	135
Representing a Vocabulary Word List	136
Implementing VocabularyAdapter	138
Fixed Movement Paging	140
Creating and Populating GridViewPager	141
Using CircledImageView	142
Using DelayedConfirmationView	143
Using WatchViewStub	146
Window Overscan	146
Using ConfirmationActivity	147
Example App #2: Confirmation Demo	148
Implementing Action Buttons	149
Creating a Fragment with an Action Button	151
Implementing DelayedConfirmationActivity	154
Implementing ConfirmationDemoActivity	156
Using BoxInsetLayout	159
Example App #3: Running Stats	160
Simulating Running Stats	160
Implementing RunStatsActivity	164
Example App #4: Creating a Custom Notification	167
Creating a Custom Notification	168
Stopping the Custom Notification	170
Starting the Custom Notification	171
Summary	171
Chapter 6: The Wearable Data Layer API	173
The Wearable Data Layer	173
Connecting to GoogleApiClient	174
Example #1: Establishing an Asynchronous Connection	174
A Minimal Asynchronous Connection	179
Establishing a Synchronous Connection	180

- PendingResults 181
- Example #2: Using the Node API 182
- Messages and Data 188
- Example #3: Building a Shared Counter 188
 - Implementing the Handheld Activity 190
 - Implementing the Wearable Activity 193
 - Sending Messages with the Message API 196
 - Receiving Messages with the Message API 197
 - Request Response Paradigm 198
- The Data API 201
 - Implementing the Decrement and Increment Buttons 205
- Example #4: Sending and Receiving Assets 207
 - Sending Assets with the Data API 207
 - Retrieving Assets with the Data API 211
- Summary 213
- Chapter 7: Creating Custom Watch Faces 215**
 - The Anatomy of a Watch Face 216
 - Interactive and Ambient modes 216
 - Low-bit Ambient Mode and Burn-in Protection 218
 - Square and Round Screens 220
 - The Moto 360 and the Bottom Inset 220
 - Interruption Filter 221
 - System Indicators 221
 - Configuring the Watch Face Style 222
 - Status Bar Gravity 223
 - Card Peek Mode 224
 - Ambient Card Peek Mode 225
 - Peek Opacity Mode 226
 - Background Visibility 227

Hotword Indicator Gravity	227
View Protection.....	228
Show System UI Time	229
Building a Basic Watch Face	230
Creating and Declaring a Watch Face	230
Implementing the Engine Class	233
Scaling the Watch Face to the Current Size	239
Drawing the Watch Face.....	240
Building a Watch Face from Bitmaps	245
Encapsulating Bitmaps with WatchFaceBitmapHolder	249
Data Layer Architecture	252
Implementing ConvergenceUtil.....	253
Implementing ConvergenceWearableConfigActivity	256
Implementing ConvergenceConfigActivity	259
Implementing ConvergenceWearableListenerService	262
Implementing ConvergenceWatchFaceService	264
Summary.....	272
■ Part IV: Google Glass	273
■ Chapter 8: Running Apps Directly on Glass	275
The GDK and the Mirror API.....	275
Should I use the Mirror API or the GDK?	276
Immersion and Ongoing Tasks	277
Immersion.....	277
Ongoing Tasks	278
Combined Immersion and Ongoing Task	279
Getting Started	280
Voice commands	282
Implementing an Immersion	282
Using CardScrollView	285
Playing System Sounds.....	285

Implementing Scrolling Cards	286
Providing Feedback.....	290
Summary	293
■ Chapter 9: Glass User Interface Essentials.....	295
Building Cards Styled for Glass	295
Declaring a Voice Command.....	296
Extending CardScrollAdapter.....	296
Creating the List of Scrolling Cards	298
The TEXT Layout	300
The TEXT_FIXED Layout.....	303
The COLUMNS Layout.....	304
The COLUMNS_FIXED Layout.....	306
The CAPTION Layout	307
The TITLE layout	308
The AUTHOR Layout.....	308
The MENU Layout	309
The ALERT Layout	310
The Ongoing Task Pattern	315
LiveCards Require Menus.....	315
Publishing LiveCards	315
Updating LiveCards.....	315
Low-Frequency Rendering	316
High-Frequency Rendering.....	321
Implementing a Digital Speedometer.....	330
Implementing DigitalSpeedView.....	331
Implementing DigitalSpeedMenuActivity.....	332
Implementing DigitalSpeedService	335
Implementing DigitalSpeedRenderer.....	337
Implementing RenderThread	339
Displaying Progress and Status with Slider	341
Summary.....	348

■ Chapter 10: Gesture and Voice Recognition	349
 Gestures on the Touchpad	349
Viewing Generic Motion Events	350
Using GestureDetector	354
Detecting Gestures and Counting Fingers	355
Detecting Scrolling on the Touchpad	358
Implementing Inertial Scroll	364
 Head Gestures	366
Detecting Head Nods	368
Implementing HeadGestureDetector	369
Implementing HeadGestureActivity	371
 Voice Recognition	374
Voice Command Prompts	375
Standalone Voice Recognition	376
Contextual Voice Commands	377
 Summary	379
■ Chapter 11: The Camera: Taking Pictures and Recording Video	381
 Taking Pictures and Recording Videos with Intents	382
Taking Pictures	382
Recording Videos	383
 Example #1: Capturing Media with Intents.....	383
Define a Voice Command.....	384
 Example #2: Using the Camera API	391
Implementing CameraUtils	392
Implementing the SurfaceHolder Callback	394
Implementing CameraActivity.....	396
Initialize the Layout and SoundPool.....	397
 Example #3: Using the Camera API from a LiveCard	402
Create LiveCameraPreview.....	402
Creating a Menu for the LiveCard.....	404
Implement CameraLiveCardService	406

Example #4: Recording a Time-lapse Video	407
Implementing TimelapseActivity.....	408
Creating an Options Menu	410
Recording a Time-lapse.....	412
Summary.....	415
■ Part V: Android Wear and Glass	417
■ Chapter 12: Location and Orientation.....	419
Modeling the Earth as an Ellipsoid.....	419
Geodetic Latitude and Longitude.....	420
Defining Altitude.....	423
Location Providers.....	423
Obtaining Location Updates on Android Wear	424
Register for Node and Location Updates	427
Using FusedLocationProvider	429
Requesting and Removing Location Updates	430
Obtaining Location Updates on Glass.....	432
Request and Remove Location Updates	435
Representing Orientation	437
Coordinate Frames	438
Azimuth, Pitch, and Roll.....	440
Rotation Matrices and Quaternions	442
Measuring Orientation.....	443
The Rotation Vector Sensor	444
Rotation Vector in Android Wear	444
Rotation Vector in Glass.....	445
Implementing Rotation Vector in Android Wear	445
Using Rotation Vector on Glass	450
Building a Compass	455
True North vs. Magnetic North.....	456
Magnetic Interference	456

Building a Compass in Android Wear	456
Create a View That Draws a Compass	457
Implementing CompassActivity	459
Update the Azimuth and Display Warnings	462
Obtaining the Declination	464
Building a Compass in Glass	465
Implementing CompassView	466
Obtain the Declination	471
Using Step Counter in Android Wear	472
Summary	475
Index	477

About the Author



Andres Calvo has designed, developed, and evaluated interfaces for wearable and mobile devices for over five years. Andres currently works for Ball Aerospace as a contractor at Air Force Research Laboratory, and he is fascinated by the potential that wearables have to provide unprecedented user experiences. During his free time, Andres develops applications for Android, Glass, and Android Wear both for fun and as a freelancer. He has received bachelor degrees in computer science and computer engineering from the University of Dayton. In Fall 2015, Andres will pursue a graduate degree, and his research will focus on creating seamless and natural wearable interfaces. Contact him at andresacalvo@gmail.com and visit his portfolio and blog at <http://andrescalvo.com> and <http://ocddevelopers.com>, respectively.

About the Technical Reviewer



Jeff Tang has successfully developed mobile, web, and enterprise apps on many platforms. He became a Microsoft Certified Developer and a Sun Certified Java Developer last century; had Apple-featured, top-selling iOS apps in the App Store; and was recognized by Google as a Top Android Market Developer. Jeff has a master's degree in computer science with an emphasis on artificial intelligence and believes in lifelong learning. He loves playing basketball (he once made 11 three-pointers and 28 free throws in a row), reading Ernest Hemingway and Mario Puzo, and fantasizing about traveling around the world.

Acknowledgments

Working with the team at Apress has been nothing but a pleasure. Android Wear and Google Glass have advanced at a remarkably fast pace, and keeping this book up to date required many unplanned revisions and schedule changes. Mark Powers, the book's coordinating editor, has adeptly managed every aspect of the development of the book while adapting to all these last-minute changes. Jane Hosie-Bounar and Lindsay Beaton have provided so much insightful feedback not just on the book's writing, but also on its technical content.

A big thank you to Jeff Tang, who gave me an initial glimpse on the book writing process when I was a technical reviewer for his book (*Beginning Google Glass Development*... check it out!). Jeff is great to work with and, now that we switched roles for this book, his feedback has drastically improved the quality of the writing and the example code. I also thank Steve Anglin, Michelle Lowman, Anamika Panchoo, and everyone else at Apress who helped make this book real, including all the people who work behind the scenes.

To the Android Wear and Glass teams at Google, thanks for helping me find what I'm passionate about. The impact of your vision and your work will only continue to expand.

Finally, perhaps with 90% seriousness, to Keurig, Flying Pizza, and Biscoff cookies—without you, I wouldn't even be halfway done.